Voice AI System Architecture Design Documentation

Executive Summary

This architecture leverages an agent-based design for a Voice AI platform, utilizing React for a responsive frontend, Flask with SocketIO for a real-time backend, and a modular agent system to ensure low latency (<1s turn) and scalability (1,000 sessions). The design employs streaming audio, caching for common responses, and compact memory summaries to meet performance targets. Trade-offs include local Whisper for STT privacy versus potential cloud speed, and Flask's simplicity versus more robust scaling options. Challenges included high latency from DeepSeek and OpenAI APIs via Azure, and resource constraints limited full efficiency, though improvements are possible with better resources.

System Overview and Goals

The Voice AI platform enables natural, multi-turn conversations via phone calls or web audio clients, converting voice to text, interpreting it with an AI, and responding with synthesized speech. Goals include low latency (first reply <500ms, full turn <2s), scalability to 1,000 concurrent sessions, and reliability with early barge-in support. The system uses streaming audio, caches frequent responses, and stores compact conversation summaries to maintain context efficiently.

Agent Descriptions

1. **Session Gateway**: Manages client connections and sessions.

   Inputs: Client audio stream;

   Process: Establishes session (100ms), tracks state;

   Outputs: Session ID, audio to Listener; Failure: Retry connection.

2. **Listener & End-pointing Agent**: Detects voice activity and preprocesses audio.

   Inputs: Audio chunks;

   Process: VAD, noise suppression (50ms);

   Outputs: Preprocessed audio; Failure: Fallback to raw audio.

3. **Speech-to-Text Agent**: Converts audio to text.

   Inputs: Preprocessed audio;

Process: Incremental transcripts with Whisper (200ms);

Outputs: Partial/final transcripts; Failure: Retry with default vocab.

4. **Orchestration Agent**: Coordinates data flow.

   Inputs: Transcripts, context;

   Process: Merges transcripts, issues requests (100ms);

   Outputs: Prompts, metrics; Failure: Timeout fallback.

5. **Context & Memory Agent**: Manages conversation context.

   Inputs: History;

   Process: Summarizes, retrieves facts (150ms);

   Outputs: Compact context; Failure: Use empty context.

6. **LLM Reasoning Agent**: Generates AI responses.

   Inputs: Prompt with context;

   Process: Response generation with DeepSeek/OpenAI (500ms+ due to latency);
   Outputs: Structured response; Failure: Cached response.

7. **Text-to-Speech Agent**: Synthesizes audio.

   Inputs: Text response;

   Process: Streaming synthesis with gTTS (300ms);

   Outputs: Audio out; Failure: Fallback to text.

8. **Analytics & Quality Agent**: Monitors performance.

   Inputs: Metrics;

   Process: Collects error rates, latency (50ms);

   Outputs: Flags; Failure: Log only.

9. **Coordinator for Scale**: Manages scaling.

Inputs: Queue depths, metrics;

Process: Triggers scaling, load shedding (200ms);

Outputs: Decisions; Failure: Default scaling.

Development Approach and Implementation Plan

Phased Plan:

- **Phase 1 - Requirements and Traffic Modeling**: Model 1,000 concurrent sessions, target 500ms latency. Verify with traffic simulation using Flask load tests.
- **Phase 2 - Prototype and Proof of Concept**: Develop React frontend with WebSocket audio, Flask backend with SocketIO, and demonstrate streaming and barge-in using Whisper and gTTS.
- **Phase 3 - Core Architecture Implementation**: Implement all agents, integrate caching with Redis, and mitigate cold starts with pre-loaded models.

Services Selection and Justification

1. **Telephony/Audio Transport**: Flask SocketIO with WebRTC (low-latency streaming, multi-tenant routing via session management).
2. **Streaming Speech Recognition**: Whisper (local deployment, high accuracy, custom vocab support).
3. **Language Model**: DeepSeek/OpenAI via Azure GitHub (throughput vs. high latency trade-off, attempted smaller models).
4. **Streaming Speech Synthesis**: gTTS (first audio latency, mid-speech interruption support).
5. **Conversation Summaries**: Redis (fast retrieval, compact storage for context).
6. **Relational/Object Storage**: PostgreSQL (sessions, billing), S3 (recordings, reliability).
7. **Compute Platform**: Flask on Gunicorn (simplicity, scales with worker processes).

Communication Patterns and APIs

- **Real-Time Bidirectional Audio**: WebSocket via Flask SocketIO, sends 100ms audio chunks with barge-in flags to handle interruptions.
- **Incremental Transcripts**: STT emits partial transcripts to Orchestration, timed at 50ms intervals for real-time updates.
- **Example**: Client → Session Gateway (audio via WebSocket) → STT → Orchestration → TTS → Client (audio response).

Platform Challenges and Proposed Solutions

- **STT Latency**: Use Whisper tiny model; Solution: Local processing, 200ms budget.
- **Scaling Limits**: Flask single-threaded nature; Solution: Gunicorn workers, load balancing.
- **Memory Overhead**: Full conversation histories; Solution: Redis for summaries.
- **API Failures**: Slow DeepSeek/OpenAI APIs; Solution: Retry with fallback cache, though latency persists.
- **Barge-In**: Interrupt handling; Solution: SocketIO event interrupts.
- **Resource Constraints**: Limited fast AI APIs found; Solution: Local optimization attempted, efficiency improves with resources.

End-to-End Latency Strategy and Timing Budget

- **Strategy**: Streaming audio, caching, parallel agent execution; challenged by slow AI APIs.
- **Timing Budget**: STT 200ms + LLM 500ms+ (due to API latency) + TTS 300ms = >1s total turn; first audio out <500ms with partial transcripts.
- **Approach**: Optimize Whisper, pre-cache responses, use async Flask tasks; latency issue unresolved without faster APIs.

Scalability Plan and Operational Considerations

- **Initial Capacity**: 1,000 sessions with Gunicorn workers and Redis.
- **Extension**: Auto-scaling with cloud instances (e.g., AWS), priority lanes for high-value clients.
- **Operations**: Monitor with Analytics, log errors with Flask logging, schedule maintenance.

Summary of Trade-offs and Open Questions

- **Trade-offs**: Local STT (privacy, latency) vs. cloud STT (speed, cost); Flask (simplicity) vs. Kubernetes (robustness); DeepSeek/OpenAI APIs (free via GitHub) vs. unacceptable latency.
- **Open Questions**: Handling diverse accents in STT; optimal worker count for Flask; availability of faster AI APIs.
- **Notes**: Significant time was spent searching for fast AI APIs without success, leading to use of DeepSeek/OpenAI via Azure, which increased latency. Resource limitations hindered 100% efficiency, but with adequate resources (e.g., faster APIs, cloud compute), efficiency could improve significantly.