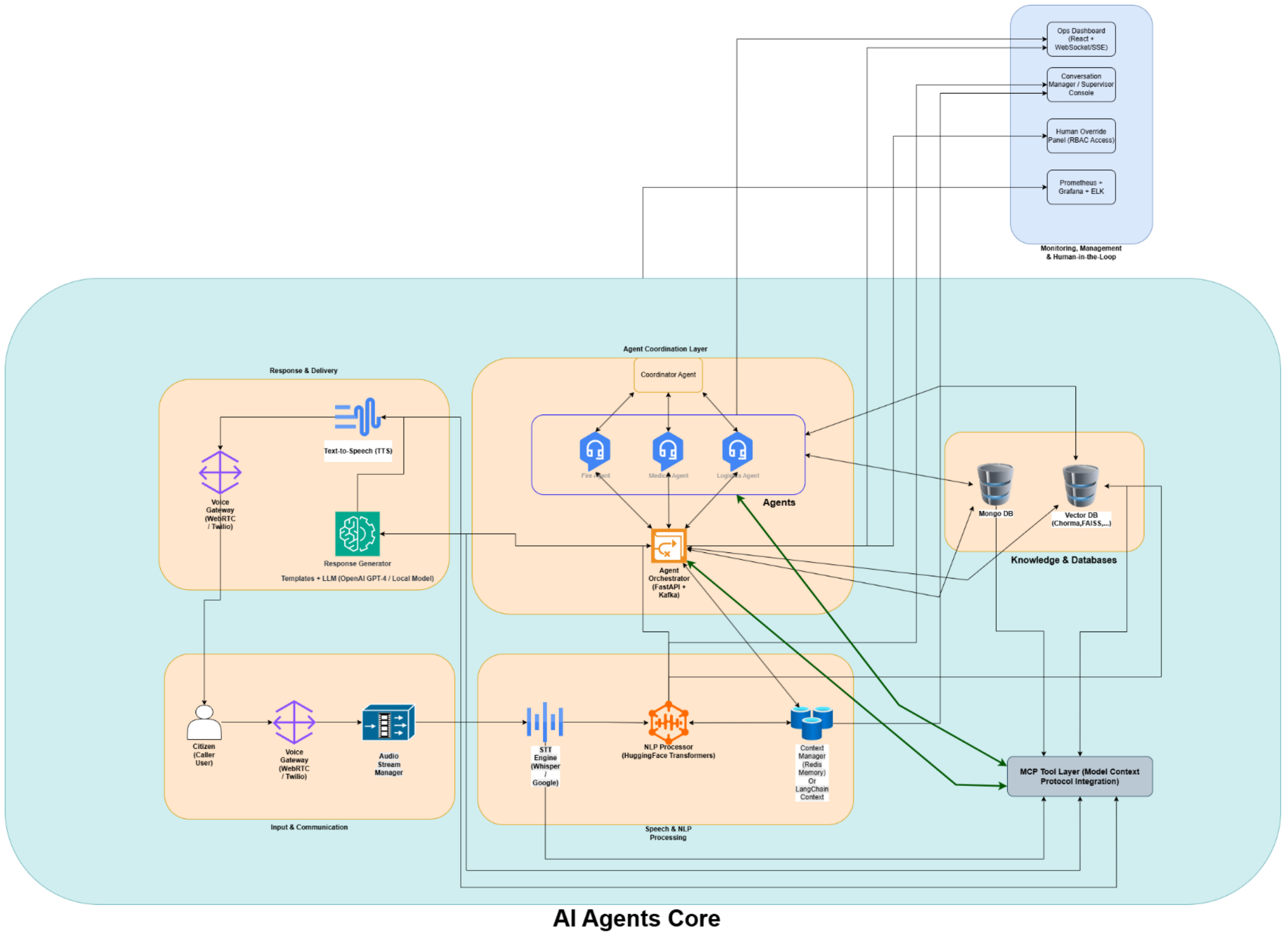
System Design of RescueHub AI Agent

Director: Arian Ganji

1. AI Agent Architecture Diagram


Ops Dashboard (React + WebSocket/SSE)
Conversation Manager / Supervisor Console
Human Override Panel (RBAC Access)
Prometheus + Grafana + ELK
Monitoring, Management & Human-in-the-Loop
Response & Delivery
Text-to-Speech (TTS)
Voice Gateway (WebRTC / Twilio)
Response Generator
Templates + LLM (OpenAI GPT-4 / Local Model)
Agent Coordination Layer
Coordinator Agent
Fire Agent
Medical Agent
Logistics Agent
Agents
Agent Orchestrator (FastAPI + Kafka)
Mongo DB
Vector DB (Chorma,FAISS,...)
Knowledge & Databases
Citizen (Caller User)
Voice Gateway (WebRTC / Twilio)
Audio Stream Manager
Input & Communication
STT Engine (Whisper / Google)
NLP Processor (HuggingFace Transformers)
Context Manager (Redis Memory) Or LangChain Context
Speech & NLP Processing
MCP Tool Layer (Model Context Protocol Integration)
AI Agents Core

2. Overview of AI Agent Architecture Diagram

The entire design of RescueHub, a real-time AI-powered emergency coordination platform, is depicted in this architecture.

The system orchestrates multiple specialized agents, transforms live citizen voice reports into structured, actionable data, and provides the caller with real-time spoken confirmations.

There are six major layers to it:

1. Input & Communication: WebRTC or Twilio are used to manage audio streams and record live audio for low-latency, real-time speech input.
2. NLP models (Hugging Face Transformers) and contextual memory (Redis/LangChain) interpret meaning and sustain multi-turn conversation flow, while speech and NLP processing uses Google STT or Whisper to transcribe speech.
3. Agent Coordination Layer: Manages Fire, Medical, and Logistics Agents through a central Coordinator Agent and FastAPI/Kafka Orchestrator, enabling collaboration across domains.
4. Knowledge & Databases: Combines MongoDB for structured incident and resource data with a Vector Database (Chroma/FAISS) for semantic retrieval and similarity-based knowledge search.
5. Response & Delivery: Uses an LLM-based Response Generator with predefined templates to compose accurate responses, and a TTS module to produce natural spoken confirmations for citizens.
6. Monitoring & Human-in-the-Loop: Includes real-time dashboards, conversation supervision, override control, and full system observability using Grafana/ELK.

Additionally, the architecture integrates the **Model Context Protocol (MCP)** as a unified tool layer connecting agents with external resources—such as databases, routing engines, and TTS services—via standardized tool calls.

MCP ensures decoupled, secure, and auditable interactions between AI agents and operational systems, allowing RescueHub to easily extend or replace external services without changing the agent logic.

Together, these layers and the MCP layer fulfill all core requirements: real-time voice processing, multi-turn conversation handling, intelligent agent collaboration, contextual knowledge retrieval, and dynamic spoken response delivery.

3. The Toolkit for Development

RescueHub uses a modular AI architecture that makes use of contemporary frameworks for multi-agent orchestration and real-time multimodal processing:

1) Input and Communication: Node.js or FastAPI for stream management and signaling; WebRTC or Twilio APIs for bi-directional low-latency audio streaming.
2) NLP and Speech Processing: Speech-to-Text: Google Cloud STT or OpenAI Whisper for instantaneous transcription.
3) NLP & Intent Recognition: Using spaCy and Hugging Face Transformers (BERT, RoBERTa) to extract and classify entities.
4) Context Memory: Redis Streams or LangChain Memory for preserving conversational states.
5) Agent Collaboration: Orchestrator: A FastAPI microservice that communicates directly through gRPC or through Kafka.
6) Agents: Logistics, Medical, and Fire Agents were refined using LLMs (OpenAI GPT-4 / local Falcon models) on domain-specific datasets.
7) Databases and Knowledge: MongoDB for operational data that is structured (resources, locations, and incidents) And Vector Database (Chroma, FAISS, or Pinecone) for semantic retrieval and similarity search.
8) Reaction and Delivery: GPT-4 and templates are used by the LLM-based Response Generator to create structured confirmation messages.
9) Text-to-Speech (TTS): For natural voice synthesis, use OpenAI Realtime, Azure Speech, or Amazon Polly.
10) Observation and Management: WebSocket/SSE for dashboards and React And Prometheus + Grafana + ELK for metrics, logs, and tracing is the observability stack.

4. Performance in Real Time

In order to preserve sub-second latency during interactions involving multiple turns:

1) Streaming Architecture: Text and audio are processed incrementally using streaming natural language processing and frame-based STT.
2) gRPC/HTTP2 Connections: Handshake delays are minimized by persistent low-overhead channels between modules.
3) Async Event Loop: To prevent blocking calls, async I/O is used by all microservices.
4) Edge Caching: Redis is used to cache frequently visited routes and incident templates.

5) Parallel Pipelines: Kafka or async queues are used by STT, NLP, and Orchestrator to function simultaneously.
6) Lightweight TTS Models: For quicker response playback, with an average of less than 300 ms.

5. Cooperation of Agents

For intelligent task routing and escalation, RescueHub uses a Coordinator Agent:

1) Incidents are routed to specialized agents (Fire, Medical, Logistics) by the Agent Orchestrator.
2) The Coordinator Agent combines signals from several agents in complex emergencies (such as a fire with injuries).
3) Agents interact with one another through Kafka topics (agent.medical.evt, agent.fire.cmd) or shared APIs.
4) Agent synchronization is ensured by a shared MongoDB incident graph and context space.
5) Multi-agent negotiation is supported by the system, allowing agents to cooperatively propose or confirm resource dispatch.

6. Management of Context

A hybrid memory framework is used to preserve context:

1) Short-Term Memory: The Discussion on Redis or LangChain For quick recall, the buffer saves the last N turns.
2) Long-Term Memory: Semantic retrieval using vector embeddings of transcripts kept in a vector database.
3) Context Injection: The top K pertinent embeddings are dynamically retrieved and combined into the prompt for every NLP or LLM call.
4) State Transitions: Conversation stages (waiting location, verifying severity, dispatch ready) are tracked by a small FSM (Finite State Machine).
5) This enables agents to stay coherent, follow up rationally, and refrain from asking the same questions over and over.

7. Effectiveness

A number of tactics guarantee economical and efficient use of resources:

1) Dynamic Scaling: STT/NLP pods automatically scale according to the volume of active calls.
2) Model Routing: Larger LLMs only get involved when necessary, while lightweight models manage straightforward interactions.
3) Batching and Debouncing: To reduce API overhead, STT segments are combined prior to NLP inference.
4) Memory Reuse: Redis's context storage with TTL prevents needless database queries.
5) Data compression and stream chunking: To save bandwidth, audio and transcripts are streamed in small frames.

8. Challenges

Challenge	**Strategy**
Vague or incomplete reports	NLP agents ask clarifying questions using a context-aware FSM; fallback to keyword + semantic similarity lookup in Vector DB.
Overlapping calls or concurrent incidents	Event-driven architecture with queue isolation per session; Redis locks prevent context collision.
Panicked callers or emotional speech	Sentiment and tone analysis guide the system to switch to a calming scripted voice or transfer to a human operator.
Background noise / speech errors	Whisper + VAD (Voice Activity Detection) filtering and confidence scoring before passing to NLP.
Agent conflict or duplicate actions	Coordinator Agent resolves overlap through incident priority rules in MongoDB.