

High level architecture

User ↔ LiveKit (WebRTC audio) ↔ Backend (STT → LLM / RAG + Action layer / MCP → TTS) ↔ LiveKit → User.

Additionally: datastore (product/catalog + orders), vector DB for RAG, observability & security layers, optional human escalation via LiveKit.

Components — what they do and why they help

1. LiveKit (server + SDKs)

- **What:** Open source realtime media server & managed offering for WebRTC, audio/video, track APIs and ingestion/egress. Provides rooms, participants, tracks, server hooks and audio sources/sinks.
- **Why helpful:** Handles the heavy realtime WebRTC plumbing (low-latency audio capture/playback, multi-participant rooms, recording, injecting synthetic audio). You don't reimplement signaling, NAT traversal, media encoding/decoding, or stream mixing — LiveKit does it. [LiveKit docs](#)

2. LiveKit SDKs (Web, iOS, Android, Node, Python)

- **What:** Client libraries for connecting to rooms, publishing/subscribing audio tracks, programmatic audio sources (push audio frames), and server APIs. Quickstarts and voice AI examples exist.
- **Why helpful:** Speed up frontend & backend integration: e.g., web app captures mic and publishes, backend can inject TTS audio as a track or capture participant audio for processing. LiveKit's `AudioSource` lets you push generated TTS audio into the room.

3. Speech-to-Text (STT)

- **What:** Convert user's speech -> text. Options: Whisper (local/hosted), cloud STT (OpenAI realtime, Google, Azure), or LiveKit's built-in transcription pipeline.
- **Why helpful:** LLMs work over text; accurate low-latency transcripts are necessary for natural dialogue. Choose latency vs accuracy tradeoff (streaming vs batch). LiveKit can integrate with streaming STT for continuous recognition.

4. LLM (core conversational brain)

- **What:** GPT (OpenAI), Claude, local LLMs (Ollama) used to generate replies, manage dialogue state, parse intent, and call tools/actions. Optionally run LLM with function calling or an agent framework.
- **Why helpful:** Handles NLU/NLG: answering product questions, recommending items, phrasing clarifying Qs. Use model choice based on latency, cost, privacy needs.

5. Retrieval-Augmented Generation (RAG) + Vector DB

- **What:** Index product catalog, FAQs, order history, policies into vector store (Pinecone, Redis, Weaviate, Milvus). On each query retrieve relevant context to ground LLM answers.
 - **Why helpful:** Prevent hallucinations by grounding responses in live product/order data — essential for accurate order tracking and factual product details.
6. **MCP (Model Context Protocol) — optional but highly recommended**
- **What:** A protocol / server approach that standardizes how LLMs request actions and access structured context (user profile, orders, tools). MCP offers a secure, auditable interface for LLM→tool interactions.
 - **Why helpful:** Instead of ad-hoc “tool-calling” code for each model, MCP acts as the canonical “tool broker” that executes operations (check order status, create return, fetch inventory) and returns typed structured data. It reduces N×M integration complexity and improves governance/auditability.
7. **Text-to-Speech (TTS)**
- **What:** Convert LLM text -> audio. Options: ElevenLabs, OpenAI Realtime TTS, LiveKit TTS injection. Output audio is pushed back to LiveKit as a track.
 - **Why helpful:** Low latency, high quality TTS gives a natural voice. LiveKit's APIs let you stream TTS audio directly into the same room so the user hears the bot in real time.
8. **Backend glue (Node.js / Python)**
- **What:** Orchestrates: receive LiveKit audio (or recording), call STT, send live transcripts to LLM + RAG retriever, call MCP to perform actions, send LLM output to TTS, and push audio back to LiveKit.
 - **Why helpful:** Central event loop and API surface for your platform. Also handles auth, logging, rate limiting, and reliability.
9. **Datastores & Services**
- **Catalog DB / Search** (Elastic/Algolia).
 - **Orders / CRM / Inventory** (your backend or third-party).
 - **Vector DB** for embeddings.
 - **Observability** (logs, traces, metrics), **quality monitoring** (MOS, WER, transcription confidence).
10. **Human escalation / agent fallback**
- **What:** If intent = “speak with human”, use LiveKit to join a human customer-support agent into the same room.
 - **Why helpful:** Improves success for complex issues; LiveKit supports multi-party rooms and recording.
11. **Security & Compliance**
- TLS, authentication for LiveKit rooms, PII handling, consent dialogs, selective transcript retention, and audit logs for MCP calls. MCP helps enable safer, auditable tool calls.

3) End-to-end runtime flow (step-by-step)

Below is a concrete runtime sequence for one user conversation turn.

1. **User joins** your site/app → LiveKit room (Web SDK). Client publishes microphone audio track.
 2. **Backend subscribes** to that participant's audio (or LiveKit forwards audio to your STT pipeline). Option A: push raw frames into Whisper/streaming STT. Option B: use a LiveKit transcription add-on or cloud STT.
 3. **STT (streaming)** returns incremental transcript (with word timestamps & confidence). Backend streams transcript to the LLM/Agent.
 4. **Agent decision / LLM**: The LLM consumes transcript + short session context (recent turns) + retrieved docs from Vector DB (RAG). If LLM determines it needs to take an action (e.g., check order, fetch product stock), it issues an action request. Use either:
 - LLM function-calling pattern, or
 - an agentic framework (LangChain/AutoGen/MetaGPT) to orchestrate tool calls.
 5. **MCP server** receives the action request (e.g., `get_order_status(order_id)`) from the agent/LLM and executes against your order DB or 3rd party (with auth), returning typed results. MCP logs the call for audit.
 6. **LLM / Agent** consumes results, composes a human-friendly reply. Optionally RAG ensures product facts are correct.
 7. **TTS** converts text → audio. Backend pushes audio frames to LiveKit (via `AudioSource` or publish new track). The user hears the bot reply in the same room.
 8. **Loop**: continue streaming STT for the next user utterance. Record metadata, metrics, and conversation transcript to analytics. If the LLM requests an outbound action (`create_return`, `apply_discount`), MCP executes and returns confirmation to be spoken to the user.
-

4) Where RAG and vector search fit

- Every time the LLM receives a user message, first call an embedder (OpenAI/CLIP/other) on the message and query the vector DB to fetch top-k relevant docs (product specs, policy, order notes). Attach those to the LLM prompt (or pass as context via RAG pipeline). This reduces hallucinations.
-

5) Agentic frameworks — can you use them? (Yes) + recommended options & flow

Why use an agentic framework? They let the LLM plan multi-step tasks and invoke tools reliably (e.g., look up order → call returns tool → summarize → confirm). Good for multi-step flows like returns, refunds, product recommendation pipelines.

Example agent flow (detailed):

1. **Trigger:** Transcript arrives and intent classified as “track order”.
2. **Planner (LLM agent)** composes sub-tasks: (a) authenticate user, (b) fetch order by ID, (c) check shipment status, (d) format reply.
3. **Tool calls:**
 - `mcp.get_user_profile(user_id)` → returns permissions & recent orders.
 - `mcp.get_order_status(order_id)` → returns order state + ETA.
 - If order not found, agent calls `mcp.create_order_investigation(order_id)` and notifies agent.
4. **RAG retrieval:** if user asks about return policy for that product, agent fetches the product policy doc and includes it in the response.
5. **Commit & audit:** agent instructs MCP to perform final operations (e.g., open a return), MCP records audit trail.
6. **Response:** LLM composes message, TTS is produced and streamed.