High-level architecture (tech stack + workflow)

Meet-joining AI assistant (EN + HI), scoped exactly to your needs.

**System at a glance**

* **Capture**: Join Google Meet and capture live audio
* **ASR**: Real-time transcription (English + Hindi, code-switching)
* **NLP**: Ask questions later (“what are the tasks?” “decisions?”)
* **Store + Search**: Time-aligned utterances with speaker labels
* **Serve**: Minimal web UI + API

**1) Feature-to-Component Mapping**

**A. Live meeting capture (Google Meet)**

**MVP (fastest, reliable)** – **Browser Extension capture**

* Chrome/Brave extension that:
  + Detects a Meet tab, uses chrome.tabCapture to grab tab audio
  + Streams PCM/Opus over WebSocket to your backend
  + Works when *you* are in the meeting (no bot account needed)
* Pros: quickest path, less brittle than headless Meet
* Cons: requires your browser to be open

**Milestone 2 (bot-like join)** – **Headless “companion” joiner**

* Headless Chromium with Puppeteer/Playwright logs in a service account, joins Meet as a silent participant
* Audio capture via **WebRTC getUserMedia** → pipe to backend (Linux server often easiest using a virtual audio device like PulseAudio null sink)
* Pros: autonomous joins
* Cons: more engineering + maintenance; Meet has no official bot API

Start with the **extension**. Add the headless joiner later.

**B. Real-time ASR (English + Hindi)**

* **Model**: Whisper **large-v3** or **faster-whisper** (CTranslate2) for low-latency, multilingual, code-switch handling
* **Pre-processing**: WebRTC VAD (py-webrtcvad) to chunk, 16kHz mono
* **Streaming**: Sliding windows (e.g., 2–3s) with *segment stitching* to reduce latency but keep accuracy
* **Post-processing**: punctuation, casing; Hindi Devanagari vs Hinglish detection (keep original where possible)

**C. Speaker labeling (optional at first, then improve)**

* **MVP**: “Speaker 1/2/3” using Meet events (if accessible) or simple turn-taking heuristics
* **Improve**: Server-side diarization with **pyannote.audio** (offline re-pass) synced to timecodes

**D. Storage + retrieval**

* **Primary DB**: PostgreSQL
  + meetings(id, title, start\_ts, end\_ts, meet\_url),
  + speakers(id, meeting\_id, label, name\_nullable),
  + utterances(id, meeting\_id, speaker\_id\_nullable, t\_start\_ms, t\_end\_ms, text)
* **Search/RAG**:
  + Build embeddings over **utterances** (or 10–30s chunks) → **FAISS** (local) or Chroma
  + Keep metadata: meeting\_id, t\_start\_ms, speaker\_id

**E. Question answering / extraction**

* **Patterns you need**:
  + “What tasks were assigned?” → extract (assignee?, task, due\_date?)
  + “Decisions?” → list decisions, owner
* **Approach**:
  + **Retrieval**: semantic search → top-k utterances
  + **LLM extractors** with **structured output** (JSON schema)
    - Use a light instruction model (e.g., **Llama-3.1-8B Instruct** via **Ollama**) or your preferred API
    - Add **regex/heuristics** as a fallback (“please”, “can you”, “we will”, “action”, “todo”, “deadline”, Hindi equivalents like “करना है”, “तय हुआ”)
* **Tables** (optional materialization):
  + tasks(id, meeting\_id, span\_ids[], text, assignee\_nullable, due\_nullable, evidence\_utterance\_ids[])
  + decisions(id, meeting\_id, text, evidence\_utterance\_ids[])

**F. Summaries & insights (second milestone)**

* Fast meeting summary (bullets), tasks, decisions
* Keep **traceability**: provide time-stamped evidence links to utterances

**G. API + UI**

* **Backend**: FastAPI
  + POST /ingest/audio (WebSocket): receives frames from extension
  + GET /meetings/:id/transcript (paginated utterances)
  + POST /qa {meeting\_id, question} → {answer, evidence}
  + GET /meetings/:id/summary → {bullets, tasks[], decisions[]}
* **UI (minimal)**: React (or plain Next.js)
  + Live transcript stream
  + Search box
  + QA panel (“tasks?”, “decisions?”)
  + Click evidence → jump to timestamp

**2) Data Flow (end-to-end)**

1. **User joins Meet** → starts extension
2. **Extension** captures tab audio → WebSocket → backend
3. **Backend** runs VAD → **faster-whisper** streaming → emits partial + final segments
4. Segments **persist** (utterances) with timestamps; optional diarization re-pass after call
5. **Embedding index** updated incrementally for new utterances
6. **User query** (“tasks?”) → retrieve top-k utterances → LLM extractor → return structured JSON + evidence

**3) Tech choices (battle-tested + local friendly)**

* **Capture**: Chrome/Brave Extension (Manifest V3, tabCapture)
* **Transport**: WebSocket (binary Opus or raw PCM16)
* **ASR**: **faster-whisper** (GPU on your RTX 3060; low latency)
* **VAD**: webrtcvad
* **Diarization**: pyannote.audio (optional re-pass)
* **Backend**: FastAPI + Uvicorn
* **DB**: PostgreSQL + SQLAlchemy
* **Vector**: FAISS (flat or IVF depending on scale)
* **LLM**: **Llama-3.1-8B Instruct** via **Ollama** (good local baseline)
  + If you later need better Hindi/code-switch extraction, consider fine-tuning a small **Qwen2.5** or **Gemma 2** variant or use task-specific prompts with examples.
* **Infra**: Single Linux box with CUDA 12.1 (you already have this stack locally); Dockerize later

**4) Schemas (minimal)**

-- meetings

id UUID PK

title TEXT

meet\_url TEXT

start\_ts TIMESTAMPTZ

end\_ts TIMESTAMPTZ

-- speakers

id UUID PK

meeting\_id UUID FK

label TEXT -- "Speaker 1"

name TEXT NULL -- optional mapped name

-- utterances

id UUID PK

meeting\_id UUID FK

speaker\_id UUID NULL

t\_start\_ms INT

t\_end\_ms INT

text TEXT

lang TEXT NULL -- "en", "hi", "mixed"

-- (optional) tasks

id UUID PK

meeting\_id UUID FK

text TEXT

assignee TEXT NULL

due\_date DATE NULL

evidence\_utterance\_ids UUID[] -- provenance

-- (optional) decisions

id UUID PK

meeting\_id UUID FK

text TEXT

evidence\_utterance\_ids UUID[]

**5) Prompting patterns (extraction)**

**Instruction summary→tasks→decisions (JSON only)**

You are extracting structured outcomes from meeting text.

Return ONLY valid JSON with keys: summary[], tasks[], decisions[].

Each item MUST include "evidence": list of {utterance\_id, t\_start\_ms}.

Input Context: <top-k utterances with timestamps>

Schema:

{

"summary": ["..."],

"tasks": [

{"task": "...", "assignee": "name|null", "due": "YYYY-MM-DD|null", "evidence": [...]}

],

"decisions": [

{"decision": "...", "evidence": [...]}

]

}

**Hindi + English note**: provide 1–2 in-prompt exemplars mixing Hinglish so the model learns code-switch patterns.

**6) Latency plan (targets)**

* Capture → backend: < 50 ms
* VAD window: 320 ms frames, 1–2s commit
* ASR (faster-whisper, GPU): ~ real-time at 1.0–1.5x for 16 kHz
* Partial captions to UI every ~500–800 ms; “finalized” lines after 1.5–2.5 s

**7) Privacy & control (pragmatic)**

* Local or self-hosted inference (no enterprise compliance for now)
* **Toggle recording** per meeting; delete meeting data endpoint
* Store only text + timestamps (skip raw audio unless you need re-processing)

**8) Milestones (practical)**

**Milestone 0 – Setup (1–2 days)**

* FastAPI skeleton, Postgres, FAISS
* faster-whisper running on GPU

**Milestone 1 – MVP capture + live transcript (3–5 days)**

* Chrome extension tabCapture → backend WebSocket
* Streaming ASR → live transcript UI
* Store utterances with timestamps

**Milestone 2 – Q&A on transcripts (2–3 days)**

* Embedding + retrieval
* LLM extractor with JSON output
* /qa endpoint + UI

**Milestone 3 – Summaries & basic diarization (2–4 days)**

* Summary endpoint
* Offline pyannote diarization re-pass
* Map segments to speakers

**Milestone 4 – Bot join (optional)**

* Headless Chromium joiner, auth, audio piping on Linux

**9) Gotchas & tips**

* **Meet audio ducking**: make sure extension captures **tab audio**, not mic; disable echo cancellation in constraints to get clean stream.
* **Code-switch**: keep **no forced language** in Whisper; let it auto-detect; persist lang per utterance if helpful.
* **Diarization drift**: re-align diarized segments to ASR timecodes with a small tolerance window.
* **Traceability**: always return evidence (utterance IDs + timestamps) with any AI answer.

**🖥️ Programming Languages**

* **Python** → backend, transcription, NLP/LLM integration
* **JavaScript / TypeScript** → browser extension, frontend UI

**⚙️ Core Tools & Frameworks**

**Frontend / Client**

* **Chrome Extension** → meeting audio capture (via chrome.tabCapture)
* **React (or Next.js)** → simple web UI (live transcript, query box, results display)

**Backend**

* **FastAPI** → REST + WebSocket backend (ingest audio, serve transcripts, QA endpoints)
* **Uvicorn** → ASGI server

**Audio & Transcription**

* **WebRTC VAD** (webrtcvad Python package) → detect speech chunks
* **Whisper / faster-whisper** (CTranslate2 backend) → real-time multilingual ASR (English + Hindi, code-switch)
* **Pyannote.audio** (optional) → speaker diarization

**Data & Storage**

* **PostgreSQL** → structured storage of meetings, utterances, tasks, decisions
* **SQLAlchemy** → ORM layer
* **FAISS** (local vector DB) → semantic search for QA over transcripts

**LLM / NLP**

* **Ollama** (running locally) with **Llama 3.1–8B Instruct** → extract tasks, decisions, summaries
* Optionally **Qwen2.5** or **Gemma 2** small models (for Hindi-heavy conversations)
* **LangChain / LlamaIndex (optional)** → if you prefer a framework for retrieval + orchestration

**🔧 Developer / Infra**

* **Docker** → packaging and portability
* **NVIDIA CUDA 12.1 + PyTorch** → GPU acceleration for Whisper & diarization (your RTX 3060)
* **GitHub** → version control
* **VS Code** → development environment

**📦 Supporting Libraries**

* **Pydantic** → request/response validation in FastAPI
* **WebSockets (Python + JS)** → live audio streaming
* **Matplotlib / Plotly (optional)** → analytics visualization (who spoke how much, etc.)
* **dotenv** → config/credentials management

✅ **Summary**:

* **Python** (FastAPI, Whisper, FAISS, Ollama) for the backend AI pipeline.
* **JS/TS** (Chrome extension, React) for capture + UI.
* **Postgres** + **FAISS** for storage & retrieval.
* **Whisper + Ollama** for transcription + meeting Q&A.

**Repository structure**

Create these files and folders: