Building a Local AI Assistant: Your Private Jarvis

February 11, 2026 · by Mark Bartlett

Download this guide as PDF

> **Quick Answer:** The fastest path to a local AI assistant: install Ollama, run Open WebUI in Docker, and you have a ChatGPT-like assistant in 15 minutes. Add voice with Open WebUI's built-in Whisper + TTS support. Add document Q&A by uploading files to Open WebUI's RAG. Add home automation through Home Assistant's Ollama integration. The whole stack runs on an 8GB GPU. For a fully custom voice pipeline (wake word, speech-to-text, LLM, text-to-speech), combine openwakeword + faster-whisper + Ollama + Kokoro TTS — expect ~1-2 second response latency on a 12GB card.

📚 **Guides referenced:** Run Your First LLM · Open WebUI Setup · Voice Chat with Local LLMs · Local RAG · Function Calling

Cloud assistants know what you ask, when you ask it, and what files you feed them. A local assistant doesn't. Everything runs on your hardware, your data stays on your machine, and there's no monthly bill.

This guide walks you through building one, piece by piece. Each level adds a capability. Stop wherever you're satisfied — Level 1 alone gives you a working assistant in 15 minutes.

What you're building

A local AI assistant chains several components:

1. **Wake word detection** — listens for "hey Jarvis" (or whatever you pick)
2. **Speech-to-text (STT)** — converts your voice to text (Whisper)
3. **LLM** — generates the response (Ollama)
4. **Text-to-speech (TTS)** — reads the response aloud (Kokoro or Piper)
5. **RAG** — searches your documents for context before answering
6. **Tools** — web search, calculations, file operations, API calls
7. **Home control** — turns lights on, checks sensors, runs automations

You don't need all seven. Most people want 1-4. The guide is structured so each level builds on the previous one.

Platform comparison

Several tools bundle parts of this stack. Here's what each handles:

	Open WebUI	AnythingLLM	Jan	LM Studio	DIY Python
Chat interface	Yes (web)	Yes (desktop/ web)	Yes (desktop)	Yes (desktop)	You build it
Voice input/ output	Yes (built-in)	No	No	No	Yes (manual)
Document RAG	Yes (built-in)	Yes (best RAG)	No	No	Yes (manual)
Function calling	Yes (BYOF editor)	Yes (agents)	Yes (MCP)	Yes (agent API)	Yes (manual)
Home automation	No	No	No	No	Yes (manual)
Multi-model switching	Yes	Yes	Yes	Yes	Yes
Setup difficulty	1 Docker command	Desktop installer	Desktop installer	Desktop installer	Write code

The recommendation: Open WebUI for levels 1-4. It handles chat, voice, RAG, and basic tools out of the box. Switch to AnythingLLM if you need better document management (workspaces, permissions, more chunking options). Go DIY Python only for the voice pipeline or home automation — those need custom code regardless.

Hardware requirements

Your assistant runs multiple models at once: the LLM, the Whisper model for speech recognition, the TTS model, and (if using RAG) an embedding model.

VRAM budget breakdown

Component	VRAM	Notes
7B LLM (Q4)	~4-5 GB	Qwen 2.5 7B, Llama 3.1 8B
14B LLM (Q4)	~8-10 GB	Qwen 2.5 14B — needs 12GB+
Whisper large-v3	~1.5 GB	Loaded only during speech input
Whisper turbo	~0.8 GB	Faster, slightly less accurate
Kokoro TTS	~0.3 GB	82M parameters, runs on CPU too
Piper TTS	~0.1 GB	Lighter, lower quality
Embedding model	~0.3-0.5 GB	nomic-embed-text or similar

What each GPU tier can run

GPU	LLM	Voice	RAG	Simultaneous?
8 GB (RTX 3060 Ti, 4060)	7B Q4	Whisper turbo + Kokoro	Yes	Tight but works. Close other apps.
12 GB (RTX 3060, 3080 12GB)	7B Q4 comfortably	Whisper large-v3 + Kokoro	Yes	Room to spare. Sweet spot.
16 GB (RTX 4070 Ti Super)	14B Q4	Full Whisper + any TTS	Yes	Everything fits with room left over.
24 GB (RTX 3090, 4090)	14B Q6 or 32B Q4	Everything	Yes	Run it all without thinking about VRAM.
CPU only (32GB+ RAM)	7B Q4 (~3-5 tok/s)	Whisper on CPU (~4-8x realtime)	Yes	Works, just slow. Usable for text chat.

Ollama handles model loading and unloading automatically. If Whisper and the LLM don't fit in VRAM simultaneously, Ollama swaps them — adds a second or two of latency but doesn't crash.

→ Use our Planning Tool to check exact VRAM for your setup.

Level 1: Text chat (15 minutes)

If you already have Ollama installed, this takes 2 minutes.

Install Ollama and pull a model

```
# Install Ollama (if not already)
curl -fsSL https://ollama.ai/install.sh | sh

# Pull a model
ollama pull qwen2.5:7b
```

Qwen 2.5 7B is a good all-rounder for assistant tasks — strong at following instructions, good at reasoning, and small enough for 8GB VRAM. Llama 3.1 8B is the other solid choice.

Start Open WebUI

```
docker run -d -p 3000:8080 \
  --add-host=host.docker.internal:host-gateway \
  -v open-webui:/app/backend/data \
  --name open-webui \
  --restart always \
  ghcr.io/open-webui/open-webui:main
```

Open `http://localhost:3000`, create an account (local only, no cloud), select your model, and start chatting. You now have a private ChatGPT-like interface.

For detailed setup (custom configs, GPU passthrough, connecting to remote Ollama), see the Open WebUI setup guide.

Level 2: Voice input and output (30 minutes)

Two paths: use Open WebUI's built-in voice, or build a standalone pipeline.

Option A: Open WebUI voice (easier)

Open WebUI has voice chat built in. In Settings > Audio:

1. **STT engine** — set to "Web API" (uses your browser's speech recognition) or configure a local Whisper endpoint
2. **TTS engine** — set to "Web API" (uses browser TTS) or point to a local TTS server

For fully local voice, configure Open WebUI to use a local Whisper server:

```
# Run faster-whisper as a server
pip install faster-whisper
# Or use the whisper-server Docker image
docker run -d -p 8765:8765 \
  fedirz/faster-whisper-server:latest-cuda
```

Then set the STT URL in Open WebUI's audio settings.

The browser-based option works immediately but sends audio to your browser's speech API (Google/Apple). For true privacy, run Whisper locally.

Option B: Standalone voice setup

For a dedicated voice assistant (not browser-based), you need three pieces running independently. Our voice chat guide covers this in detail. The short version:

```
pip install faster-whisper  # Speech-to-text
pip install kokoro          # Text-to-speech (~82M params)
# Ollama is already running from Level 1
```

Latency breakdown (RTX 3060 12GB, 7B model):

Stage	Time
Whisper turbo (STT)	~0.3-0.5 sec
LLM first token	~0.2-0.4 sec
LLM full response	~1-3 sec (depends on length)
Kokoro TTS (first audio)	~0.1-0.3 sec
Total to first spoken word	**~0.8-1.5 sec**

That's comparable to Alexa's response time, though the LLM's answer takes longer to finish speaking than a canned Alexa response.

Level 3: Ask questions about your files (30 minutes)

RAG (Retrieval Augmented Generation) lets your assistant search your documents before answering. Feed it PDFs, notes, code, emails — it finds the relevant sections and includes them in the LLM's context.

Our RAG guide covers three setup methods in depth. Here's the fastest.

Open WebUI RAG (built-in)

Pull an embedding model:

```
ollama pull nomic-embed-text
```

In Open WebUI, go to Workspace > Knowledge, create a collection, and upload your files. Open WebUI handles chunking, embedding, and retrieval. When you chat, toggle the knowledge base on for that conversation.

Supported formats: PDF, TXT, DOCX, CSV, Markdown, and code files.

AnythingLLM (better for large document sets)

If you have hundreds or thousands of documents, AnythingLLM handles it better. It gives you workspaces (separate document collections), more chunking options, and a no-code agent builder.

```
docker pull mintplexlabs/anythingllm
docker run -d -p 3001:3001 \
  -v anythingllm:/app/server/storage \
  mintplexlabs/anythingllm
```

Point it at your Ollama instance, upload documents, and chat. AnythingLLM uses the same embedding models as Open WebUI but gives you more control over how documents are split and searched.

What to expect

RAG works well for:

- Answering questions about specific documents ("What does section 4.2 of this contract say?")
- Searching across large collections ("Which meeting notes mentioned the Q3 budget?")
- Code Q&A ("How does the authentication module work in this repo?")

RAG struggles with:

- Summarizing entire books (context window too small for full content)
- Questions that need reasoning across many documents simultaneously
- Highly structured data (use a database, not RAG)

Level 4: Give it tools (1 hour)

A chat assistant can only answer from its training data and your documents. Tools let it take actions — search the web, run calculations, check APIs, read live data.

Our function calling guide covers the protocol in detail. Two ways to add tools:

Open WebUI tools (no code)

Open WebUI has a built-in Python function editor (BYOF — Bring Your Own Function). Go to Workspace > Tools, click "Create Tool," and write a Python function. Open WebUI handles the function-calling protocol with the LLM automatically.

Example — a web search tool:

```
import requests

class Tools:
    def search_web(self, query: str) -> str:
        """Search the web for current information.

        :param query: The search query
        :return: Search results as text
        """
        # Use SearXNG or another local search engine
        resp = requests.get(
```

```
            "http://localhost:8888/search",
            params={"q": query, "format": "json"}
        )
        results = resp.json()["results"][:3]
        return "\n".join(
            f"- {r['title']}: {r['content']}" for r in results
        )
```

The LLM sees the function signature and docstring, decides when to call it, and Open WebUI executes the code and feeds results back.

Ollama function calling (API-level)

If you're building a custom assistant in Python, Ollama supports function calling directly:

```
import ollama

response = ollama.chat(
    model="qwen2.5:7b",
    messages=[{"role": "user", "content": "What's the weather in Denver?"}],
    tools=[{
        "type": "function",
        "function": {
            "name": "get_weather",
            "description": "Get current weather for a location",
            "parameters": {
                "type": "object",
                "properties": {
                    "location": {"type": "string", "description": "City name"}
                },
                "required": ["location"]
            }
        }
    }]
)

# Model returns tool_calls — you execute them and feed results back
```

Qwen 2.5 7B is the best function-calling model at this size — 0.933 F1 for tool selection, close to GPT-4 level. See the function calling guide for the full agentic loop pattern with error handling.

Practical tools worth adding

Tool	What it does	Complexity
Web search	Answers questions about current events	Medium (needs SearXNG or similar)
Calculator	Math without hallucination	Easy
File reader	Read local files on demand	Easy
Calendar/reminders	Check schedule, set alerts	Medium
Shell commands	Run system commands (careful with this one)	Easy but risky
Home Assistant API	Control smart home devices	Medium (see Level 5)

Level 5: Control your home (2+ hours)

This is where it gets Jarvis-like. Connect your assistant to Home Assistant and it can control lights, check temperatures, lock doors, and trigger automations.

Home Assistant + Ollama

Home Assistant has native Ollama integration since HA 2025.6. Setup:

1. Settings > Devices & Services > Add Integration > Ollama
2. Enter your Ollama server URL (e.g., `http://192.168.1.100:11434`)
3. Select a model
4. Under “Assist,” set it as your conversation agent

Now you can type or speak commands in Home Assistant and the local LLM handles them.

The home-llm approach (specialized model)

General-purpose LLMs aren’t great at home control out of the box. They don’t know your device names, and they hallucinate entity IDs. The home-llm project solves this with a 3B model fine-tuned specifically for smart home commands:

```
ollama pull fixt/home-3b-v3
```

This model understands Home Assistant's entity format and generates valid service calls. It's much more reliable for "turn off the kitchen lights" than a general 7B model, though it can't hold a general conversation.

The practical setup: use your general-purpose model (Qwen 2.5 7B) for conversation and RAG, and route home-control commands to home-3b-v3. Home Assistant's conversation pipeline supports this kind of routing.

Wyoming protocol for voice satellites

Want to talk to your assistant from every room? The Wyoming protocol lets you place voice satellites (a Raspberry Pi with a microphone and speaker) around your house, all connected to your central Home Assistant + Ollama server.

Each satellite runs:

- **openWakeWord** — listens for your wake word
- **Whisper** (via Wyoming) — converts speech to text
- **Piper TTS** (via Wyoming) — speaks the response

The LLM processing happens on your main GPU machine. The satellites just handle audio I/O.

Hardware per satellite: Raspberry Pi 4/5 (~~$35-60), a USB microphone (~~$10), and a speaker. Total per room: under $80.

Honest assessment

Home automation is the most fragile part of this stack. Some realities:

- Simple commands work reliably: "Turn off the bedroom lights," "What's the temperature downstairs," "Lock the front door."
- Complex commands are hit-or-miss: "Turn on the lights in every room except the nursery" may or may not work depending on how your entities are named.
- The LLM doesn't know your entity names unless you expose them explicitly. Expose too many and you eat context window. Expose too few and it can't help.
- Latency for voice commands through a Wyoming satellite is 3-5 seconds end-to-end. Not instant like a cloud assistant.

- The home-llm 3B model is more reliable for device control but can't handle follow-up conversation.

It works. It's private. It's also clearly a generation behind commercial voice assistants for smart home control. If you go in knowing that, you'll be fine.

The full DIY voice pipeline

If you want a standalone voice assistant (not browser-based, not Home Assistant), the script below chains everything together.

Requirements

```
pip install ollama faster-whisper kokoro sounddevice numpy openwakeword
```

Minimal voice loop

```
import ollama
import sounddevice as sd
import numpy as np
from faster_whisper import WhisperModel
from kokoro import KPipeline

# ── Init models ──────────────────────────────────────
whisper = WhisperModel("turbo", device="cuda", compute_type="float16")
tts = KPipeline(lang_code="a")  # American English

SYSTEM_PROMPT = "You are a helpful assistant. Keep responses concise — under 3 sentences when po
conversation = [{"role": "system", "content": SYSTEM_PROMPT}]

def record_audio(duration=5, sample_rate=16000):
    """Record audio from microphone."""
    print("Listening...")
    audio = sd.rec(
        int(duration * sample_rate),
        samplerate=sample_rate,
        channels=1,
        dtype="float32"
    )
    sd.wait()
```

```
    return audio.flatten()

def transcribe(audio):
    """Convert speech to text with faster-whisper."""
    segments, _ = whisper.transcribe(audio, language="en")
    return " ".join(s.text for s in segments).strip()

def speak(text):
    """Convert text to speech with Kokoro."""
    for _, _, audio in tts(text):
        sd.play(audio, samplerate=24000)
        sd.wait()

def chat(user_message):
    """Send message to Ollama, get response."""
    conversation.append({"role": "user", "content": user_message})
    response = ollama.chat(
        model="qwen2.5:7b",
        messages=conversation
    )
    reply = response["message"]["content"]
    conversation.append({"role": "assistant", "content": reply})
    return reply

# ── Main loop ──────────────────────────────────────
print("Voice assistant ready. Press Ctrl+C to stop.")
while True:
    try:
        audio = record_audio(duration=5)
        text = transcribe(audio)
        if not text or len(text) < 2:
            continue
        print(f"You: {text}")
        reply = chat(text)
        print(f"Assistant: {reply}")
        speak(reply)
    except KeyboardInterrupt:
        print("\nStopped.")
        break
```

This is a minimal version — no wake word, no silence detection, fixed recording duration. For wake word support, add openwakeword and record in a continuous loop, triggering transcription only after detecting the wake phrase. The ollama-STT-TTS project on GitHub has a more complete implementation with silence detection via webrtcvad.

Latency targets

Setup	First spoken word	Full response
8 GB GPU, 7B model	~1.5-2.5 sec	~3-6 sec
12 GB GPU, 7B model	~0.8-1.5 sec	~2-4 sec
24 GB GPU, 14B model	~0.8-1.2 sec	~2-4 sec
CPU only, 7B model	~5-10 sec	~15-30 sec

The bottleneck is the LLM, not the speech processing. Whisper turbo transcribes in under 0.5 seconds on any modern GPU. Kokoro generates audio faster than real-time on CPU alone.

What local assistants still can't do

Time for honesty. A local assistant in 2026 is genuinely useful, but it's not Alexa or Siri in several ways.

Where local wins:

- Privacy. Nothing leaves your machine. No recordings stored on corporate servers.
- No subscription. No monthly fee, no API costs, no rate limits.
- Customization. You choose the model, the system prompt, the tools, the voice.
- Offline. Works without internet (after initial model download).
- Document search. Feed it your files — something Alexa will never do.

Where local still falls short:

- Response time. 1-2 seconds vs Alexa's ~0.5 seconds for simple queries. CPU-only is much slower.
- Always-on listening. Cloud assistants run wake-word detection on dedicated low-power chips. A local setup needs a Raspberry Pi satellite or your PC running constantly.
- Ecosystem. No Spotify integration, no Amazon shopping, no thousands of "skills." You build each integration yourself.
- Multi-room audio. Wyoming satellites work but setup is manual and finicky compared to dropping an Echo in each room.
- Accuracy for home control. Cloud assistants have been trained on millions of smart-home interactions. Local models are still catching up.

The gap is closing. A year ago, local voice latency was 5-10 seconds. Now it's under 2. Models like Qwen 2.5 handle tool calling almost as well as GPT-4. But if your primary use case is "set a timer" and "play music," a $30 Echo is still better at that specific job.

The local assistant wins when your use case involves private documents, custom tools, or anything you don't want a corporation listening to.

The bottom line

Start with Level 1. Ollama + Open WebUI takes 15 minutes and gives you a private ChatGPT. Most people are surprised how capable this is on its own.

Add voice when you want hands-free interaction. Add RAG when you have documents to search. Add tools when you need live data. Add home automation if you're already running Home Assistant.

You don't need to build everything at once. Each level is useful on its own, and each one links to a detailed guide if you want to go deeper.

The full stack — voice, documents, tools, home control — runs on a single 12GB GPU. That's a $200 used RTX 3060. Your private Jarvis costs less than a year of ChatGPT Plus.

Related guides

- Run Your First Local LLM
- Open WebUI Setup Guide
- Voice Chat with Local LLMs: Whisper + TTS
- Local RAG: Search Your Documents
- Function Calling with Local LLMs
- What Can You Run on 8GB VRAM?
- What Can You Run on 12GB VRAM?

Sources: Open WebUI Docs, Home Assistant Ollama Integration, home-llm, Wyoming Protocol, Kokoro TTS, faster-whisper, ollama-STT-TTS, AnythingLLM

Get notified when we publish new guides.

Subscribe — free, no spam
