# Maximizing OpenWebUI Capabilities: Comprehensive Feature Integration

In this deep-dive, we outline how to enable **maximum capabilities** in an OpenWebUI deployment. Each section below corresponds to a key use-case or feature from the checklist, with configuration steps, considerations, and citations for reference. The goal is an all-in-one OpenWebUI stack with persistent storage, secure multi-user access, local and hosted LLMs, Retrieval-Augmented Generation (RAG), web browsing, image and audio processing, external tool integration, and more – all **out of the box**.

## Persistent Storage (Survives Restart)

OpenWebUI stores user data (conversations, uploaded files, etc.) on disk. By default this is within the container’s filesystem (e.g. a SQLite DB and file storage under /app/backend/data). To preserve data across restarts, mount a persistent volume or use external storage:

* **Volume Mount:** Use a Docker volume for /app/backend/data. For example, the official Docker run command uses -v open-webui:/app/backend/data[[1]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=settings%20panel%20to%20Ollama%20or,OpenAI%20compatible%20endpoints). This ensures chats, attachments, and configs survive container restarts.
* **External Object Storage (Optional):** OpenWebUI supports S3-compatible storage for attachments. You can set STORAGE\_PROVIDER="s3" and provide S3\_ACCESS\_KEY\_ID, S3\_SECRET\_ACCESS\_KEY, S3\_ENDPOINT\_URL, S3\_REGION\_NAME, and S3\_BUCKET\_NAME env variables[[2]](https://docs.openwebui.com/tutorials/s3-storage/#:~:text=which%20means%20Open,storage)[[3]](https://docs.openwebui.com/tutorials/s3-storage/#:~:text=To%20use%20Amazon%20S3%2C%20we,S3_BUCKET_NAME). With this, files are stored in an S3 bucket instead of the local volume. (For self-hosting, something like a Supabase storage or MinIO could be used as an S3 endpoint.)
* **What to Choose:** For simplicity, a shared volume or network drive is often sufficient for persistence[[4]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=4,a%20storage%20backend%20like%20S3). In scalable clusters, using S3 or NFS is recommended for durability[[4]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=4,a%20storage%20backend%20like%20S3).

Either way, ensure backups of the data volume or bucket are taken regularly (for example, tar the open-webui volume or configure your S3 backup). **Database persistence**: By default OpenWebUI uses SQLite; consider an external Postgres DB (DATABASE\_URL) for better reliability[[5]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=%60REDIS_SENTINEL_HOSTS%3Dredis,links%20%26%20search)[[6]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=SQL%20%28Postgres%29%20,links%20%26%20search). If using Postgres, include that in backups as well.

## Authentication Enforced (Admin + Signups Off)

For a secure setup, you likely want one admin account and disable open sign-ups for others:

* **Disable Self-Signup:** Set ENABLE\_SIGNUP=False in the environment. By default it’s True, but turning it off will remove the ability for random users to create accounts[[7]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=).
* **Admin Account:** Ensure you have an admin user. On first launch, you can sign up and that account can be promoted to admin. Alternatively, set DEFAULT\_USER\_ROLE=admin temporarily so that new users (yourself) get admin by default[[8]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=). Once your admin exists, you can turn signups off.
* **Default Role:** Another approach is leaving signups on but making all new users “pending” by default (DEFAULT\_USER\_ROLE=pending) so they require manual approval[[8]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=). However, in a single-user or closed setup, it’s simpler to just disable signups entirely.
* **Login and OAuth:** Ensure ENABLE\_LOGIN\_FORM=True (default) so the login form is available[[9]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=). If using OAuth/SSO in the future, set WEBUI\_URL properly **before** enabling it[[10]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=), but for now a local admin user is simplest.

After these settings, only the admin (or pre-created users) can access the system. This satisfies **“admin only”** usage. (Always keep note of the admin credentials; if lockout occurs with signups off, you’d have to re-enable signups via an env var or DB change to get back in.)

## Local LLM Available (Ollama / LocalAI Backend)

To use local models, we integrate a local LLM server with OpenWebUI:

* **Ollama** (for running LLaMa/CGPT models locally): Deploy the Ollama container or binary on the same Docker network. By default, OpenWebUI can connect to Ollama’s API on port 11434. Set ENABLE\_OLLAMA\_API=True to enable the Ollama integration[[11]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=). Then point OpenWebUI to the Ollama server URL. If OpenWebUI and Ollama are in Docker Compose, you can use the service name, e.g. OLLAMA\_BASE\_URL="http://ollama:11434"[[12]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=match%20at%20L538%20,Else%20%60http%3A%2F%2Fhost.docker.internal%3A11434). This allows OpenWebUI to list and use models served by Ollama.
* *Tip:* Ensure your Ollama container is running with the desired models pre-loaded or available. OpenWebUI will query it for available models at startup (it caches the model list for performance)[[13]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=and%20Ollama%20API%20endpoints%20,fresh%20API%20calls%20each%20time).
* **LocalAI** (OpenAI API-compatible local backend): If you prefer LocalAI (which exposes an OpenAI-like API for local models), you can integrate it as well. You would treat it like a hosted OpenAI endpoint: set OPENAI\_API\_BASE\_URL to LocalAI’s URL and use an API key if required. Many users find Ollama integration more straightforward for local models, but LocalAI is an option if you have it running (e.g., on http://localai:8080/v1 with OPENAI\_API\_KEY set to any token expected by LocalAI).

With either approach, after configuration, you should see the local models available in OpenWebUI’s model dropdown. For example, if using Ollama with a model named WizardLM, you can select and chat with it entirely locally.

## Hosted LLMs via OpenRouter (OpenAI-Compatible API)

OpenWebUI excels at combining local and hosted models. To use hosted LLMs (OpenAI, Anthropic, etc.) cost-effectively, we’ll configure OpenWebUI to use **OpenRouter** as a gateway:

* **OpenRouter Setup:** Obtain an API key from OpenRouter (which can route to multiple model providers). In OpenWebUI, set OPENAI\_API\_BASE\_URL="https://openrouter.ai/api/v1" (the base URL for OpenRouter’s OpenAI-compatible API)[[14]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=). Then set OPENAI\_API\_KEY to your OpenRouter API key (which typically starts with or-... if it’s an OpenRouter token). This will make OpenWebUI send OpenAI-format requests to OpenRouter instead of directly to OpenAI.
* **Multiple Keys/Models:** If you plan to use different keys or endpoints, OpenWebUI supports multiple base URLs and keys (via OPENAI\_API\_BASE\_URLS and OPENAI\_API\_KEYS) for load-balancing or multi-provider setups[[15]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=). But a single OpenRouter endpoint is usually sufficient since OpenRouter itself can route to various models.
* **Testing:** After configuring, add a model in OpenWebUI’s admin pointing to an OpenAI-model (like gpt-4 or claude-v1 if supported) using the “OpenAI API” connection type. OpenRouter will handle the request. Make sure to enable any required headers for OpenRouter (some require an X-OpenAI-Proxy header, but in most cases just the base URL and key are enough, as OpenRouter mimics the OpenAI API).
* **Cost Guardrails:** Using OpenRouter allows usage of community or cheaper endpoints, but you should still monitor usage. We’ll address rate limiting and cost control in a later section.

Now you can leverage both local and cloud models in OpenWebUI’s interface – for example, a local 13B model for casual queries and GPT-4 via OpenRouter for complex tasks.

## File Uploads (PDF/Audio/Images) Functionality

OpenWebUI supports uploading files in the chat or knowledge base, which is crucial for RAG and analysis features. We need to ensure PDF, audio, and image uploads “just work” by configuring content extraction and file size limits:

* **Content Extraction for Documents:** By default, OpenWebUI has a built-in text extractor for common formats, but it’s limited. For robust PDF support (and many other document types), enable the Apache Tika integration. Set CONTENT\_EXTRACTION\_ENGINE="tika" to use a local Tika server[[16]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=). The default TIKA\_SERVER\_URL is http://localhost:9998[[17]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=), so you should run an Apache Tika server (Docker image apache/tika:latest works well) on the same network at port 9998. With this, any PDF or DOCX you upload will be sent to Tika for text extraction. This dramatically improves file upload capabilities (Tika handles PDFs, Word, Excel, HTML, etc.). **Note:** Ensure the OpenWebUI container can reach the Tika container (set the hostname if needed; e.g. TIKA\_SERVER\_URL="http://tika:9998" if the service name is tika). OpenWebUI’s docs confirm you have these options[[18]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=RAG%20Content%20Extraction%20Engine%3A%20You,mistral_ocr) and that each extraction engine has its config requirements[[18]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=RAG%20Content%20Extraction%20Engine%3A%20You,mistral_ocr).
* **OCR for Images/PDF Scans:** If you need OCR (scanned PDFs or images with text), consider using document\_intelligence or docling engines with an OCR backend (or even Mistral OCR)[[18]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=RAG%20Content%20Extraction%20Engine%3A%20You,mistral_ocr). These require additional services or API keys. For a start, Tika can extract text from many images/PDFs (it has OCR support if Tesseract is installed, though in Docker it might be limited). For more advanced OCR, OpenWebUI can integrate with Mistral’s OCR API (MISTRAL\_OCR\_API\_KEY) or Docling server, but that’s optional.
* **Audio Files (Speech-to-Text):** OpenWebUI can transcribe audio uploads (like .mp3 or .wav). The Speech-to-Text engine by default will use OpenAI Whisper if you have an OpenAI API key, but we want offline capability. OpenWebUI actually includes a local Whisper model if you leave AUDIO\_STT\_ENGINE empty, using the default Whisper (tiny) model built-in[[19]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=,variable). To improve this:
* For **better accuracy** with local STT, you might use a Faster-Whisper server or larger model. One approach: set up AUDIO\_STT\_ENGINE="openai" and point the OpenAI base URL to an open-source Whisper API (if available). However, since we have OpenRouter configured, OpenWebUI might try to use OpenRouter for Whisper by default – which currently might not support it. Instead, consider running faster-whisper as an OpenAI-like API. If that’s complex, the simplest path is to rely on the built-in local Whisper (which will download a Whisper model on first use). No API key needed if using local engine.
* Ensure the ENABLE\_SPEECH\_TO\_TEXT toggle is on in the UI (this is usually controlled by the presence of an STT engine and WEBUI\_URL being set to an HTTPS origin for microphone use).
* *Result:* When you upload an audio file or use the microphone, OpenWebUI will transcribe using Whisper and inject the text into the chat.
* **Image Files:** Uploading images to chat does not by itself trigger a caption or OCR (OpenWebUI doesn’t natively “describe” an image unless you have a tool/pipeline for it). The image will be stored and you can reference it or ask the assistant to analyze it **if** you have an image analysis tool set up (OpenWebUI doesn’t have one by default). If image understanding is needed, you could integrate a tool like BLIP or CLIP interrogator via OpenAPI. Otherwise, images are mainly used for the assistant to possibly *output* images (for generation tasks) or as static attachments. Since the use-case here is likely generation/editing, we handle that in the next sections.

**File Size Limits:** It’s important to adjust upload limits to avoid 413 Request Entity Too Large errors. We address this in **“Large File Uploads”** below, since it involves Traefik configuration.

## Embeddings and RAG Over Uploaded Content (Vector DB Integration)

To enable Retrieval-Augmented Generation (where the assistant can recall information from uploaded documents or knowledge bases), OpenWebUI uses a vector database to store embeddings. By default it uses an in-process Chroma DB, but for larger scale and multi-container usage, an external vector DB is recommended[[20]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=High%20QPS%20vector%20search%3A%20The,pinecone):

* **External Vector DB (Qdrant):** We choose Qdrant (an open-source vector DB) for this setup. Deploy a Qdrant container (as listed in the architecture). In OpenWebUI’s env, set VECTOR\_DB="qdrant"[[20]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=High%20QPS%20vector%20search%3A%20The,pinecone). Provide the connection details: QDRANT\_URI should point to your Qdrant instance (e.g. http://qdrant:6333 if using Docker Compose with service name qdrant)[[21]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=). If Qdrant requires an API key, set QDRANT\_API\_KEY (not needed if unsecured on the LAN)[[22]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=). By using Qdrant, all embeddings of your documents will be stored externally, allowing persistence and better performance at scale.
* **High Throughput RAG:** The Medium guide notes that the in-container Chroma becomes a bottleneck under load, and suggests configuring an external vector DB like Qdrant or others for high QPS scenarios[[20]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=High%20QPS%20vector%20search%3A%20The,pinecone). We have done exactly that. Qdrant will handle similarity search for document chunks.
* **Embedding Model & API:** To embed documents, OpenWebUI either uses a local transformer or calls an API. Running large embedding models in the container is slow, so it’s recommended to use a lighter-weight or external embedding service[[23]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=Embedding%20Model%20Engine%3A%20You%20don%E2%80%99t,openai). Two good options:
* **OpenAI Embeddings API:** Since we have OpenRouter/OpenAI configured, we can set RAG\_EMBEDDING\_ENGINE="openai"[[24]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=Task%20Model%20Configuration%3A%20Open%20WebUI,your%20use%20cases%20are%20met) and choose a model like text-embedding-ada-002 (or OpenAI’s new smaller models) by setting RAG\_EMBEDDING\_MODEL accordingly[[23]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=Embedding%20Model%20Engine%3A%20You%20don%E2%80%99t,openai). For example, OpenAI’s text-embedding-ada-002 is powerful and will be used via OpenRouter.
* **Ollama Embeddings:** Alternatively, OpenWebUI supports using Ollama to generate embeddings (RAG\_EMBEDDING\_ENGINE="ollama") if your local model can produce embeddings[[25]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=%2A%20%60ollama%60%20,variable). This might be slower or less accurate depending on the local model. Given we have OpenRouter access, using OpenAI’s optimized embedding is a good trade-off.
* **Hybrid Search:** OpenWebUI can do hybrid search (combine vector similarity with keyword search) by setting ENABLE\_RAG\_HYBRID\_SEARCH=True[[26]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=,vector%20store%20to%20leverage%20reranking), which can improve results for certain queries. This uses a BM25 index plus the vector store. It’s optional but worth testing with your data.
* **Pipeline and Memory:** Once configured, any uploaded PDF or text can be indexed. Users can ask questions, and the assistant will retrieve relevant chunks from Qdrant to answer with citations. Ensure your environment has ENABLE\_WEB\_SEARCH and RAG turned on (in Admin UI, there are toggles for “Allow Document Retrieval” or similar which correspond to these env vars).

In summary, we have set up a robust RAG pipeline: documents → Tika (text) → embedding (OpenAI or Ollama) → Qdrant vector store. This allows the assistant to provide grounded answers using user-provided data.

## Web Search in Chat (Internet Access via SearxNG)

To enable the assistant to fetch up-to-date information from the web, OpenWebUI provides a **Web Search** feature. We will configure it to use a private SearxNG meta-search engine:

* **Enable Web Search:** Set ENABLE\_WEB\_SEARCH=True in the environment to turn on the web search toggle in the UI[[27]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=). This allows the user (or automated pipeline) to perform searches.
* **Search Engine Selection:** Choose SearxNG as the search backend (since we are self-hosting one). Set WEB\_SEARCH\_ENGINE="searxng"[[28]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=). OpenWebUI supports many engines (Google PSE, Brave, Kagi, etc.)[[28]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=), but SearxNG is a great choice for privacy and unlimited queries.
* **Configure SearxNG URL:** Provide the URL to your SearxNG instance’s search API. For example, if you have SearxNG running at http://searxng:8081 with the format=json API, you’d set SEARXNG\_QUERY\_URL="http://searxng:8081/search?q=<query>" (use your actual host/port and ensure q=<query> is present)[[29]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=). The docs give an example format[[29]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=). If your SearxNG requires a search API key or specific parameters, include those in the URL.
* **Web Loader Concurrency:** By default, OpenWebUI will fetch up to 3 results and crawl them concurrently. The env var WEB\_SEARCH\_RESULT\_COUNT (default 3) controls how many top results to retrieve[[30]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=). You can adjust it (e.g., to 5 or 10) if needed. Also WEB\_LOADER\_CONCURRENT\_REQUESTS (default 10) controls parallel fetching of result pages[[31]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=). The defaults are usually fine.
* **Test Search:** In the OpenWebUI chat, you should now see a 🔍 “Search the web” button or the assistant may autonomously use the search when needed (if allowed by the prompt/pipeline). Try asking a question that requires current information – the assistant should generate search queries, SearxNG will return results, and OpenWebUI will scrape the webpages to find relevant text. The assistant then cites those sources in its answer.

*Security note:* The WEB\_SEARCH\_TRUST\_ENV variable can allow using a proxy if needed (not usually required for SearxNG on same network)[[32]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=). Also, ensure your WEBUI\_URL is set to your domain if you want links in answers to use the correct base URL[[10]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=) (this is mainly for the UI to properly link citations).

By integrating SearxNG, your OpenWebUI can browse the internet in real-time, greatly expanding its capability to provide up-to-date answers[[33]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=Open%20WebUI%20is%20one%20of%2C,ecosystem%20of%20tools%20and%20functions).

## Image Generation (Text-to-Image)

We want the assistant to generate images from text prompts (txt2img), using local Stable Diffusion-based tooling (no external API costs). OpenWebUI supports integration with image generation engines:

* **Choose an Engine:** We will use **ComfyUI**, as it is a powerful workflow-based Stable Diffusion server. OpenWebUI natively supports ComfyUI and AUTOMATIC1111 as backends for image generation[[34]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=%2A%20%60comfyui%60%20,Default%3A%20%60openai). Set IMAGE\_GENERATION\_ENGINE="comfyui" in the environment to enable this[[34]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=%2A%20%60comfyui%60%20,Default%3A%20%60openai).
* **Deploy ComfyUI:** Run a ComfyUI server (the ComfyUI Docker or running it on host). For instance, use the comfyui/comfyui Docker image and expose its API (ComfyUI typically runs a GRPC/Web API on a port, or you might use an extension that provides a REST API). Ensure it’s accessible to OpenWebUI.
* **Configure API URL:** Set COMFYUI\_BASE\_URL to the URL of the ComfyUI API[[35]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=). For example, COMFYUI\_BASE\_URL="http://comfyui:8188/api" if ComfyUI’s REST API is at port 8188 (replace with actual port/path). If ComfyUI requires an API key, set COMFYUI\_API\_KEY accordingly (not usually needed for local).
* **Workflow Customization:** ComfyUI is very flexible – you can design workflows for different tasks. OpenWebUI can trigger a specific ComfyUI workflow if you provide its ID or file via COMFYUI\_WORKFLOW[[36]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=). If left default, it will likely use a basic text2image workflow. For initial setup, you can use the default or a simple workflow JSON for txt2img. Later, you might add workflows for img2img or inpainting (addressed in the next section).
* **Automatic1111 Alternative:** If you prefer AUTOMATIC1111 (Stable Diffusion WebUI API), set IMAGE\_GENERATION\_ENGINE="automatic1111" and AUTO1111\_API\_URL (env var not shown here but analogous to ComfyUI’s) to your A1111 instance’s API (e.g., http://auto1111:7860). AUTOMATIC1111 has a /sdapi/v1 txt2img endpoint. Either engine works; ComfyUI is chosen here for its flexibility and pipeline support (the Reddit setup referenced also used ComfyUI).
* **Enable in UI:** In OpenWebUI Admin -> Settings, there is likely a toggle for “Image Generation” or it may auto-enable if the env var is set. After configuration, the chat UI will show an image generation button (🖼️ icon) or you can prompt the assistant to create an image.
* **Test Generation:** Try a prompt: “Create an image of a futuristic city skyline at sunset.” The assistant should invoke the image generation function, ComfyUI will generate the image, and OpenWebUI will display the result. Verify that the image appears and no errors in logs.

With this, you have local text-to-image generation integrated. It’s fully on-prem and can leverage your GPU. (Make sure the ComfyUI container has access to a GPU if you have one, and that the model weights (e.g., Stable Diffusion 1.5 or 2.1) are loaded in ComfyUI.)

## Image Editing (Img2Img and Inpainting)

In addition to basic generation, we want the ability to modify images – e.g. provide an input image and have the AI transform it (img2img) or fill in missing parts (inpainting). Achieving this with our ComfyUI setup requires setting up appropriate workflows:

* **ComfyUI Workflows:** Design or import a ComfyUI workflow for img2img or inpainting. For example, an **img2img** workflow would take an input image and a prompt to generate variations. An **inpainting** workflow would take an input image, a mask, and a prompt. ComfyUI can do both if configured with the right nodes. Once you have a workflow (say, saved as inpaint.json in ComfyUI), you can load it via API. OpenWebUI’s COMFYUI\_WORKFLOW env var can be pointed to a workflow file or ID[[36]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=). Alternatively, you might have to switch workflows manually via the ComfyUI API using function calls from OpenWebUI (which is advanced).
* **Triggering in Chat:** OpenWebUI doesn’t have separate UI buttons for “inpaint” vs “txt2img”; it generally has a single image generation function. To utilize img2img, the user might upload an image and then instruct, e.g., “Please modify the above image to have a red sky.” If OpenWebUI passes the image to the generation function, your ComfyUI workflow needs to detect an input image is present and do img2img. In practice, this might require a custom function or pipeline to handle (the integration isn't completely plug-and-play for editing).
* **Alternative Approach (AUTOMATIC1111):** The Automatic1111 API has dedicated endpoints for img2img and inpainting. If you set up AUTOMATIC1111, OpenWebUI might allow choosing the mode via the function arguments. For example, if the user prompt contains something like “using this image,” OpenWebUI might call img2img automatically. This depends on OpenWebUI’s implementation. Keep in mind that as of now, the integration might not fully support complex image editing without some manual steps.
* **Example Use:** Once configured, test by uploading an image (say a simple drawing or photo) and ask the assistant to modify it. Monitor the ComfyUI logs to see if it received the image input. You may have to adjust the prompt or function parameters. If it’s not straightforward, an interim solution is to use the **OpenAPI Tools** feature: for example, create a tool that calls ComfyUI with an img2img workflow explicitly. This way, you can instruct the assistant to use that tool for editing tasks.

*In summary*, image editing is achievable but may require custom setup. The ComfyUI integration is powerful – you can script very advanced operations – but ensure the assistant knows when to use it. With everything set, you have a creative AI capable of not just generating art from scratch but also altering and improving given images, all within the chat interface.

## MCP Tools Enabled in Chat (Filesystem, HTTP, etc.)

OpenWebUI allows extending its capabilities with external **tools** (via OpenAPI). The user’s request for “MCP tools” refers to tools implemented using the Model Context Protocol (MCP) and exposed to OpenWebUI through an OpenAPI proxy. In practice, this means we can enable the assistant to perform actions like reading files, making web requests, querying Reddit/YouTube, etc., by plugging in community-developed tools.

**Setup of MCP Tools via mcpo:**

* The OpenWebUI team provides a utility called **mcpo** (MCP-to-OpenAPI proxy)[[37]](https://docs.openwebui.com/openapi-servers/mcp/#:~:text=This%20documentation%20explains%20how%20to,users%20and%20developers). This proxy can host multiple MCP tools as separate OpenAPI endpoints under one server. For example, you can run a single mcpo server that serves a *filesystem tool* at /fs, an *HTTP request tool* at /http, a *web scraper tool* at /webscrape, etc. This avoids managing many separate processes.
* **Deploy Tools:** Identify the tools you need. Common ones from the community include:
* **Filesystem Tool:** Allows reading/writing files on the server (useful for the assistant to retrieve file contents or save outputs).
* **HTTP Tool:** Allows making HTTP requests to arbitrary URLs (if web search is not enough or to access specific APIs).
* **Web Scraper (Browser) Tool:** Although we have web search, a direct web-scrape tool can take a URL and return page text, which can be handy for directly accessing known URLs.
* **Reddit/YouTube Tools:** As mentioned in the Reddit example, there are tools to fetch Reddit posts or YouTube video transcripts. These are essentially specialized HTTP tools.
* **Jupyter Tool:** We have Jupyter integration separately, but one could imagine an MCP tool to run code directly (if not using the built-in code interpreter).
* The **OpenAPI Tool Servers repository** by OpenWebUI contains reference implementations for many such tools (e.g., a “time” server, “memory” server, etc.)[[38]](https://docs.openwebui.com/openapi-servers/open-webui/#:~:text=Open%20WebUI%20v0,powered%20tool%20servers). Clone that repository and look at the servers/ directory for available tools. For instance, a filesystem tool might be present there.
* **Running mcpo:** You can launch mcpo via Python (install it with pip install mcpo). For example, to run multiple tools: mcpo --port 8000 -- mcp-server-fs --root=/app/data :: mcp-server-http (this is a pseudo-command illustrating running two tools). The syntax allows chaining commands with ::. The OpenWebUI docs on MCP show using uvx to run multiple tools in one go[[39]](https://docs.openwebui.com/openapi-servers/mcp/#:~:text=uvx%20mcpo%20,your_mcp_server_command)[[40]](https://docs.openwebui.com/openapi-servers/mcp/#:~:text=You%20can%20easily%20run%20the,OpenAPI%20proxy%20%28%60mcpo%60%29%20like%20this).
* Once mcpo is running (say on port 8000), you’ll have an OpenAPI server that auto-documents these tools. OpenWebUI can connect to it: go to **Admin Settings -> Tools -> Add Tool Server**. Add the URL http://mcpo:8000/fs (for example) as a Global tool (so all users/contexts can use it)[[41]](https://docs.openwebui.com/openapi-servers/open-webui/#:~:text=Image%3A%20Settings%20Page)[[42]](https://docs.openwebui.com/openapi-servers/open-webui/#:~:text=). Repeat for each tool subpath you have (each tool on the mcpo server will have its own OpenAPI spec at e.g. /fs/docs).
* **Using Tools in Chat:** The assistant can now invoke these tools via function calls when needed. For instance, if you ask, “Open the file budget.xlsx and summarize it,” the assistant might use the filesystem tool (if programmed) to read the file (assuming the file is in an allowed directory). Or if you say “Fetch the content of example.com and analyze it,” it can use the HTTP tool. The OpenWebUI integration with these tools is “seamless” – the assistant will choose to call them if the prompt and tool availability make it appropriate[[33]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=Open%20WebUI%20is%20one%20of%2C,ecosystem%20of%20tools%20and%20functions).

**Security Note:** Exposing filesystem or HTTP tools can be dangerous if not restricted – only enable what you need and consider limiting scope (for filesystem, perhaps restrict to a specific folder). The MCP proxy approach **wraps the tools in a standard HTTP API** with documentation, making them easier to manage securely[[43]](https://docs.openwebui.com/openapi-servers/mcp/#:~:text=,specific%20glue%20code)[[44]](https://docs.openwebui.com/openapi-servers/mcp/#:~:text=,Happier%20developers%20%26%20users%20%E2%9C%85).

After configuration, test each tool individually: e.g., via the OpenWebUI UI, use the “Tools” menu or simply ask the assistant to perform a known task. Ensure responses make sense (the assistant will output the tool’s result or a summary). This system of tools greatly extends what the assistant can do – essentially plugging in “plugins” much like ChatGPT plugins, but all self-hosted.

## Command Palette Shortcut – “New Research Project” (n8n Workflow)

The user mentioned a *Command Palette -> “New Research Project”* action that triggers an n8n workflow. This implies a custom integration: using OpenWebUI’s UI to quickly launch an external automation (n8n is a workflow automation tool).

While OpenWebUI doesn’t natively know about your n8n, we can achieve this as follows:

* **n8n Setup:** Ensure the n8n container is up and reachable (in the architecture, n8n is at n8n.valuechainhackers.xyz:5678). Create a webhook in n8n that will initiate whatever “New Research Project” workflow you desire. Copy the webhook URL (it might be something like https://n8n.yourdomain/webhook/new-project).
* **OpenWebUI Command Palette:** OpenWebUI has a command palette (usually opened by Ctrl+K) for quick actions. Out-of-the-box it has things like creating a new chat, switching chats, etc. It doesn’t directly have a way to call external URLs. However, you can extend OpenWebUI by adding a **custom function button or pipeline**:
* One approach: Use the **Function** integration. You could create a dummy OpenAPI tool that calls the n8n webhook (essentially an HTTP call). But invoking it from a palette may be tricky.
* Simpler: Create a special **prompt or pipeline** in OpenWebUI that when triggered, gives a link or triggers the workflow. For example, you could make a “function” that returns a URL and have the user click it. This is not as neat as direct integration but is a workaround.
* Another approach: If you are comfortable editing OpenWebUI’s frontend code, you could add a custom action to the palette that opens a new browser tab to the given URL or sends a request (this requires forking/modifying the UI).
* **Intended Workflow:** The idea of “New Research Project” might be to automate project setup (perhaps provisioning a new Jupyter notebook, creating a folder in Nextcloud, etc., via n8n). So even a link that opens n8n or sends a signal could suffice.
* **Execution:** As an interim solution, you can instruct: *“Type /newproject to start a research project.”* If the user does that, the assistant (or a pipeline) could intercept that command and call n8n’s webhook. For instance, use a **Pipelines filter** that looks for the command and performs an HTTP call (Pipelines can run arbitrary Python).
* Using Pipelines: Write a small filter pipeline in Python that recognizes a user message like “/newproject” and then uses requests to POST to the n8n webhook, then maybe injects a reply confirming the project is created. You can load this pipeline into OpenWebUI (similar to how the Langfuse pipeline is added)[[45]](https://docs.openwebui.com/tutorials/integrations/langfuse/#:~:text=In%20the%20Admin%20Settings%2C%20create,API%20with%20the%20following%20details)[[46]](https://docs.openwebui.com/tutorials/integrations/langfuse/#:~:text=Step%204%3A%20Adding%20the%20Langfuse,Filter%20Pipeline).
* Then, you wouldn’t even need the command palette per se; the user could just input the slash command in chat.
* **Admin UI Configuration:** Check if Admin -> Settings -> **Command Palette** or **Actions** allows adding custom links. Some chat UIs allow adding custom actions that appear in the UI. If OpenWebUI doesn’t expose it, the pipeline method above is your best bet.

In summary, while not plug-and-play, it’s feasible to have a one-click or one-command trigger in OpenWebUI that signals n8n. This highlights the extensibility of OpenWebUI – with a bit of scripting, it can integrate with external automation tools to orchestrate complex workflows (e.g., setting up a research environment, sending notifications, etc.).

*(No direct doc citation for this, as it is a custom integration plan.)*

## Jupyter Notebook Integration for Code Execution (Python “Jupyter Coding”)

OpenWebUI introduced a **Code Interpreter** feature (similar to ChatGPT’s) which allows the assistant to execute code. We want to link this to JupyterHub/Notebook so that code runs in a real environment:

* **Jupyter Server:** We have a JupyterHub (or Jupyter Notebook) service running (architecture shows Jupyter at 10.0.6.15:8000 for Hub or 8888 for notebook). For simplicity, consider a single-user Jupyter Notebook server (e.g., a minimal-notebook Docker on port 8888 with a token). Ensure it’s up and the token is known (e.g., set JUPYTER\_TOKEN=123456 as in the example)[[47]](https://docs.openwebui.com/tutorials/jupyter/#:~:text=jupyter%3A%20image%3A%20jupyter%2Fminimal,JUPYTER_TOKEN%3D123456)[[48]](https://docs.openwebui.com/tutorials/jupyter/#:~:text=Open,jupyter).
* **Enable Code Interpreter:** In OpenWebUI env, set ENABLE\_CODE\_INTERPRETER=True[[48]](https://docs.openwebui.com/tutorials/jupyter/#:~:text=Open,jupyter). This turns on the code execution capability.
* **Configure Jupyter Backend:** Set CODE\_EXECUTION\_ENGINE="jupyter" (this tells OpenWebUI to use a Jupyter backend for code exec)[[48]](https://docs.openwebui.com/tutorials/jupyter/#:~:text=Open,jupyter). Then provide the Jupyter details:
* CODE\_EXECUTION\_JUPYTER\_URL pointing to the Jupyter server. Since OpenWebUI is in Docker, and if Jupyter is another container, use the Docker network address. For instance, http://jupyter:8888 (if the service name is jupyter). If running JupyterHub, use the appropriate user’s notebook endpoint. The example uses host.docker.internal which is relevant if Jupyter was on the host machine[[48]](https://docs.openwebui.com/tutorials/jupyter/#:~:text=Open,jupyter).
* CODE\_EXECUTION\_JUPYTER\_AUTH="token" and CODE\_EXECUTION\_JUPYTER\_AUTH\_TOKEN="123456" (the token or password)[[48]](https://docs.openwebui.com/tutorials/jupyter/#:~:text=Open,jupyter). These authenticate OpenWebUI to Jupyter. In our case, we use the preset token from the Jupyter container.
* Also mirror these for CODE\_INTERPRETER\_... variables, if present, as shown in the config[[48]](https://docs.openwebui.com/tutorials/jupyter/#:~:text=Open,jupyter). Essentially, the “Code Interpreter” and “Code Execution” might be separate settings; to be safe, apply to both.
* **Install Kernels:** The Jupyter minimal-notebook comes with a Python kernel. If you want R support, you could install an R kernel in that environment. The OpenWebUI integration as of v0.5.16 is primarily aimed at Python (the feature was inspired by ChatGPT’s Python sandbox)[[49]](https://docs.openwebui.com/tutorials/jupyter/#:~:text=Jupyter%20Notebook%20Integration). However, since Jupyter supports R, **in theory** you could run R code if the assistant is instructed to and the R kernel is default or specified. This is advanced – likely, for now, assume Python only through this interface.
* **Test Python Execution:** In the chat, ask the assistant to do something like “Calculate 5! in Python and plot a graph of y=x^2 from 0 to 5.” The assistant should recognize it needs to use the code tool, write Python code, execute it on the Jupyter backend, and return the output (and graph if any). You should see it utilize the Jupyter container (check logs or the Jupyter interface – it may actually create a live notebook).
* **Persistent Notebooks:** One cool aspect: since it’s using Jupyter, the code state might persist across interactions (until that kernel is restarted). This means you can do step-by-step coding. Also, you can open the Jupyter notebook in a browser (at the Jupyter URL) to inspect what the assistant is doing – it’s literally executing in a notebook environment, which is great for transparency.
* **Resource Limits:** Jupyter will execute code, so make sure the container has appropriate resources. You might also want to mount a volume for Jupyter (so any files created by code persist, or to allow file tool access to them). In our docker-compose, we saw a jupyter\_data volume for the work directory[[50]](https://docs.openwebui.com/tutorials/jupyter/#:~:text=jupyter%3A%20image%3A%20jupyter%2Fminimal,JUPYTER_ENABLE_LAB%3Dyes).

This integration turns OpenWebUI into a powerful coding assistant, offloading execution to a robust Jupyter environment. Data science, plotting, file manipulations, etc., become possible in conversation. (The user specifically requested “Jupyter coding from chat,” and this fulfills that requirement.)

## R Execution from Chat (RStudio / R Integration)

R integration is not as direct as Python, but we have a couple of options to allow R code execution in the chat:

* **Via Jupyter:** As noted, Jupyter can support R kernels. If you install an R kernel (e.g., using IRkernel in the Jupyter container), you might coax OpenWebUI to run R code. The current OpenWebUI code interpreter doesn’t automatically switch languages; it assumes the code is Python unless perhaps you explicitly specify another language. If a user says “Here is some R code: ...”, the assistant might treat it as a generic code block but attempt to run it with Python (which would error). There’s no official multi-language support yet in the UI.
* One way to handle it: If R usage is important, you could run a separate code execution engine for R. For example, set up an **RServe** or **HTTP API** for R, and add it as a tool (similar to how we added other tools). Then instruct the assistant to use that tool when R is requested. This requires custom tooling (an R HTTP server).
* **RStudio Server:** The architecture includes RStudio (10.0.5.17). RStudio is an IDE and doesn’t directly provide an API to run code on demand (aside from its console which is user-interactive). So leveraging RStudio in OpenWebUI isn’t straightforward unless you manually copy code.
* **Simplest Approach:** Encourage use of Python for most things (since we have that working), and treat R more manually. If needed, you (the user) could open RStudio for heavy R tasks. However, if you want the AI to assist with R, you can still have it **write R code** and either:
* Copy-paste that into RStudio yourself, or
* Use the filesystem tool to save the R code to a file and then run it via a custom script. For instance, have an MCP tool that executes shell commands: then the assistant could call a tool to run Rscript mycode.R after writing the file. This is hacky but possible.
* **Future Possibility:** If OpenWebUI’s code interpreter feature evolves to support multiple languages, it might allow selecting R. The Jupyter integration *could* be extended by the team to allow specifying a kernel per code block (not currently in v0.6, but maybe down the line).

In summary, **R from chat is not turn-key**. A pragmatic solution: treat it as out-of-scope for automated execution, but the assistant can still help **write R code** and you execute it externally. Since the user specifically listed R, I acknowledge it: with more development (like adding an R kernel and possibly instructing the assistant “use R kernel”), it could be done. For now, focus on Python where we have full support[[51]](https://docs.openwebui.com/tutorials/jupyter/#:~:text=Jupyter%20Notebook%20is%20an%20open,thought%20process%20and%20findings%20along) (Jupyter’s popularity in data science covers Python and Julia primarily, with R supported but less common in these AI assistant contexts).

*(No direct citations, as R integration is mostly conceptual here. However, [50†L89-L97] notes Jupyter supports languages like R, indicating the theoretical path.)*

## Voice Input (Speech-to-Text via Whisper)

OpenWebUI supports **voice input**, allowing you to speak to the assistant. We will enable this using Whisper for transcription:

* **Microphone in UI:** Ensure your OpenWebUI instance is served over HTTPS (browsers allow microphone access only on secure contexts or localhost). Traefik with TLS is already set up, so https://open-webui.yourdomain should be available. This satisfies the requirement to use mic input.
* **Speech-to-Text Engine:** By default, OpenWebUI will use the browser’s SpeechRecognition if available. But to use Whisper (OpenAI’s model) for accuracy and multi-language, we configure server-side STT:
* For offline usage, leave AUDIO\_STT\_ENGINE blank (or explicitly whisper). The docs say an empty value defaults to the built-in local Whisper inference[[19]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=,variable). OpenWebUI comes with a small Whisper model (tiny or base) and will download it on first use. This might be slower for long audio, but it’s private.
* If you have a GPU and want faster transcription, you can run a **Faster Whisper** server or use OpenAI’s Whisper API. If using OpenAI’s, set AUDIO\_STT\_ENGINE="openai" and ensure OPENAI\_API\_KEY is set (we did for OpenRouter)[[52]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=). Note that OpenRouter might not proxy the Whisper endpoint; check OpenRouter’s docs. If not, you could set AUDIO\_STT\_OPENAI\_API\_BASE\_URL separately back to OpenAI just for STT, or run local.
* Another engine option is **Deepgram** or **Azure** (supported via deepgram or azure settings)[[19]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=,variable)[[53]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=), but those are external services. We’ll stick to Whisper for now.
* **Faster-Whisper (Optional):** If transcription is too slow, you can deploy faster-whisper as an API (there are Docker images that expose a HTTP server for Whisper). Then treat it like OpenAI: set AUDIO\_STT\_ENGINE="openai" but point AUDIO\_STT\_OPENAI\_API\_BASE\_URL to your whisper server’s URL, and put a dummy AUDIO\_STT\_OPENAI\_API\_KEY if needed (or adjust if the server expects no auth). This tricks OpenWebUI into using that endpoint. This is an advanced tweak, but it’s workable.
* **Using Voice:** In the chat UI, you’ll see a microphone icon (🎤) in the input field once voice input is enabled. Clicking it will start recording and then transcribe. Alternatively, you can upload audio files (as mentioned earlier) and they will be transcribed the same way.
* **Verify:** Test by clicking the mic and saying something. The transcribed text should appear and then the assistant responds. Check logs for Whisper usage. If using local whisper, the WHISPER\_MODEL environment can control model size (default is tiny). Upgrading to base or small model might improve accuracy at cost of speed – you can download a larger Whisper model manually and point to it if needed.
* **VAD (Voice Activity Detection):** There is a setting WHISPER\_ENABLE\_VAD if you want the system to auto-stop recording when silence is detected[[54]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=,variable). This is convenient for not having to click to end recording. You can enable it if not already true.

With Whisper integrated, you have a fully voice-enabled assistant. Speak your queries and have it respond (even with voice output, as we’ll do next). It’s like running your own voice assistant akin to Siri or Alexa, but with the power of ChatGPT and custom tools.

## Voice Output (Text-to-Speech via Piper/Coqui)

To complete the voice interface, we set up **text-to-speech (TTS)** so the assistant can talk back. The user specifically wants offline TTS (Piper or Coqui), so we’ll integrate a local TTS engine:

* **Built-in vs Custom TTS:** OpenWebUI has two main TTS methods out of the box:
* Using the **browser’s SpeechSynthesis API** (Web Speech API) – this is the default if you leave AUDIO\_TTS\_ENGINE empty[[55]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=,variable). It uses the client-side voice, which is fast and requires no setup, but voices depend on the user’s device and are not always great.
* Using external TTS engines like Azure, ElevenLabs, or OpenAI’s TTS. We want neither internet nor cost, so we won’t use ElevenLabs (needs API key) or Azure.
* A “transformers” engine is listed, which presumably uses a local model from HuggingFace Transformers for TTS[[56]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=%2A%20%60elevenlabs%60%20,Speech%20engine%20to%20use). However, OpenWebUI’s documentation on that is sparse; it might refer to using something like Coqui-TTS models via the Transformers library, but it likely requires specifying which model and ensuring the environment can handle it.
* **Piper (OpenTTS) Integration:** Piper is a fast, CPU-efficient TTS (from Coqui) that can generate speech with various voices. There is a community solution to integrate Piper with OpenWebUI: **openedai-speech** by Matatonic[[57]](https://docs.openwebui.com/tutorials/text-to-speech/openedai-speech-integration/#:~:text=info). This is a server that exposes an OpenAI-compatible /v1/audio/speech endpoint locally, effectively tricking OpenWebUI into thinking it’s using OpenAI’s new TTS API, while actually using Piper under the hood. We will use this:
* Deploy the openedai-speech Docker (choose the -min variant for Piper-only if you have no GPU). The compose and Dockerfile are provided in the GitHub repo[[58]](https://docs.openwebui.com/tutorials/text-to-speech/openedai-speech-integration/#:~:text=You%20can%20use%20any%20of,the%20following%20Docker%20Compose%20files)[[59]](https://docs.openwebui.com/tutorials/text-to-speech/openedai-speech-integration/#:~:text=). Once running (say on port 5002, as an example), you have a local endpoint for TTS.
* Configure OpenWebUI TTS to use OpenAI engine: AUDIO\_TTS\_ENGINE="openai"[[55]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=,variable) and set AUDIO\_TTS\_OPENAI\_API\_BASE\_URL="http://openedai-speech:5002/v1" (assuming that’s the endpoint) and an AUDIO\_TTS\_OPENAI\_API\_KEY (which isn’t actually needed by openedai-speech, but OpenWebUI will require something – you can set a dummy value)[[60]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=).
* What happens: when the assistant needs to speak, OpenWebUI will POST to openedai-speech with the text, expecting an audio stream back. The openedai-speech server uses Piper voices to synthesize and returns audio. Piper can handle multiple languages and has high quality voices, all offline[[61]](https://docs.openwebui.com/tutorials/text-to-speech/openedai-speech-integration/#:~:text=match%20at%20L249%20,TTS%20also%20supports%20multilingual%20voices).
* *Note:* In the openedai-speech .env, you can choose voices. Piper supports multiple voice models; you might configure a specific voice (via voice\_to\_speaker.yaml in openedai-speech)[[62]](https://docs.openwebui.com/tutorials/text-to-speech/openedai-speech-integration/#:~:text=,speech%60%20service).
* **Coqui TTS Alternative:** If not using openedai-speech, another route is running **Kokoro TTS** server (as per OpenWebUI community tutorials). There is a “Kokoro-FastAPI using Docker” tutorial in OpenWebUI docs, which likely describes running Coqui’s TTS server and selecting “Custom TTS” in the UI[[63]](https://github.com/open-webui/open-webui/discussions/12937#:~:text=Introducing%20Custom%20TTS%20Engine%20Support%21,Custom%20TTS%20API%20Key%3A). Recently, OpenWebUI added **Custom TTS Engine** support in the UI where you can specify an API endpoint and key for any TTS engine[[64]](https://www.reddit.com/r/LocalLLaMA/comments/1dfa02r/openaicompatible_tts_wrapper_ie_for_openwebuis/#:~:text=feature%29%20www,TTS%20Settings%20and%20connect%20it). This might simplify integration (you’d select “Custom TTS” in Admin -> Audio settings, and input the openedai-speech endpoint and a dummy API key). Check the latest UI – if “Custom TTS” is an option, use that with openedai-speech’s URL.
* **Testing TTS:** Once configured, in a chat, you should see a speaker icon 🔊 or similar. After the assistant generates a reply, clicking that should play the synthesized voice. Alternatively, there may be an auto-read option. Test by asking the assistant to read a paragraph. Listen to ensure the voice is the Piper one (you’ll recognize it if you chose a specific voice, e.g., a certain accent or gender).
* **Quality and Performance:** Piper is quite fast (real-time or better on CPU for many voices)[[65]](https://docs.openwebui.com/tutorials/text-to-speech/openedai-speech-integration/#:~:text=,1GB%20vs.%208GB). If you need even higher quality or different styles, you could experiment with Coqui’s larger models (but those might require GPU). Piper strikes a good balance with a variety of downloadable voices (many based on high-quality datasets like LibriTTS). Also, because this is local, there’s no cost per character and no data leaves your server[[57]](https://docs.openwebui.com/tutorials/text-to-speech/openedai-speech-integration/#:~:text=info).

With voice output enabled, your assistant can speak answers. Combined with voice input, you’ve effectively created a voice assistant. The user can have a full spoken conversation with the system – all processing (STT and TTS) is on-prem, ensuring privacy and low latency.

*(Citations: We cited the openedai-speech info*[*[57]*](https://docs.openwebui.com/tutorials/text-to-speech/openedai-speech-integration/#:~:text=info) *and Piper details*[*[61]*](https://docs.openwebui.com/tutorials/text-to-speech/openedai-speech-integration/#:~:text=match%20at%20L249%20,TTS%20also%20supports%20multilingual%20voices) *to affirm this setup.)*

## Evidence Graph Query & Visualization (Neo4j Integration)

For advanced use-cases like knowledge graph queries or evidence mapping, a Neo4j graph database is included. The idea is to allow the assistant to query and maybe visualize data from a **Neo4j** graph (for example, relationships in supply chain data, or an evidence graph of research papers).

Setting this up involves:

* **Neo4j Deployment:** Ensure the Neo4j container is running and accessible (architecture has it at neo4j.valuechainhackers.xyz:7474 for browser and bolt on 7687 likely). You’ll have a Neo4j database ready. Populate it with your data or connect Neo4j to where data is stored.
* **Connecting OpenWebUI:** There is no direct plugin for Neo4j in OpenWebUI out-of-the-box. We have to integrate via a custom tool or pipeline:
* **Option 1: Cypher Query Tool** – Create an OpenAPI tool that accepts Cypher queries and returns results. You could write a small Flask/FastAPI that connects to Neo4j (via py2neo or Neo4j Python driver) and executes queries. This tool can be mounted similarly to others via mcpo or standalone. The assistant can then call it like query\_graph({"cypher": "MATCH (p:Person)-[REL]->(c:Company) RETURN ..."}). This requires some coding but is straightforward if you know Neo4j’s API.
* **Option 2: Use Python interpreter** – Since we have Jupyter/Python, the assistant could use Python code to query Neo4j. For instance, the user can ask, “Run a Cypher query to find X”. The assistant (if allowed in code mode) can do !pip install neo4j (if not installed) and then use the Python neo4j driver in a code block to query. This is a bit clunky in conversation but is possible. You might want to pre-install the neo4j driver in the Jupyter container to avoid pip lag.
* **Option 3: Static export** – If visualization is the goal, maybe generate an image of a graph. For example, using Python networkx + matplotlib to draw a subgraph. The assistant could do that in code interpreter: query Neo4j for subgraph, then draw it, then output image. This again would leverage the code tool (complex, but doable in steps).
* **Visualization:** Neo4j itself has a web UI (Neo4j Bloom or Browser). If the goal is to see the graph, you might just use those outside of OpenWebUI. But if you want the assistant to provide a visualization *inside* chat, the approach is as above: have it generate a PNG of the graph and return it. This is a creative use of the code execution + image generation (not SD, but plotting).
* **Security:** Only allow the assistant to run queries you’re comfortable with. If using a tool server, you can restrict it to read-only queries for safety.

Given the complexity, you might treat this as a specialized task: when you need graph insights, either do it manually or with targeted instructions. OpenWebUI’s strength is that it *can* incorporate these with some effort, but out-of-box it won’t know your data model. You’ll have to prompt it on how to use the graph (or embed some context about how data is structured).

In short, we **can integrate Neo4j**, but it’s a custom extension. Once done, the assistant could answer questions like “Show the connection between Company A and Supplier B in the supply chain graph” by querying Neo4j and even returning a chart. This remains an **expert-level integration**; ensure all pieces (drivers, tools) are properly set up and tested.

## Quarto PDF Report Generation

For generating reports or exporting conversation results, **Quarto** can be used. Quarto is a scientific and business report generation tool (successor to RMarkdown) that can produce PDFs from markdown/notebooks. Here’s how it fits in:

* **Install Quarto CLI:** Quarto would need to be installed either on the host or inside one of the containers (Jupyter container is a candidate, since it likely has Pandoc which Quarto relies on). You can add a step in the Dockerfile to install Quarto CLI (it’s a single binary).
* **Use Cases:** Suppose after an interactive session, you want a nicely formatted PDF of the findings. You could have the assistant help compose a report in Markdown, then use Quarto to render it. This can be done via the code interpreter: for example, the assistant could save the conversation or summary as report.qmd and then run !quarto render report.qmd --to pdf. This would produce a PDF file (assuming LaTeX is available in the environment or using Quarto’s PDF engine).
* **Automating via n8n:** Alternatively, this might be part of that “New Research Project” or an n8n workflow. Maybe one step in n8n is to take content and run Quarto on it (n8n could execute a command on the server).
* **OpenWebUI UI:** There isn’t a built-in “Export to PDF” button as far as I know, but the conversation can be copied or saved as Markdown/HTML from the UI manually. Quarto integration is a “nice to have” for generating polished outputs beyond the chat.
* **Testing:** If you install Quarto, test it by creating a sample .qmd file with some content and running quarto render. Ensure all dependencies (like LaTeX if needed) are in place. Possibly use a lightweight approach (Quarto can use Chrome headless for PDF if LaTeX is not installed, via --to pdf:chrome).
* **Integration with Assistant:** You might instruct the assistant explicitly: *“Please produce a PDF report of these results.”* The assistant could then produce a markdown, and you would run Quarto manually. Full automation would require the assistant to call a tool or code to do it, which might be complicated to trigger automatically.

In summary, having Quarto available “out of the box” means the environment is prepared for generating reports. It’s not something the assistant will spontaneously use unless directed, but it’s available for you to leverage. This addresses the requirement of being able to create polished documents from the research – an important capability for presenting findings.

*(No direct citations; this is more an internal tooling suggestion. It ties into code execution and external automation rather than OpenWebUI itself.)*

## Tracing and Observability (Langfuse Integration)

Monitoring and tracing the AI’s behavior is crucial for debugging and improving. **Langfuse** is an open-source observability platform for LLM applications. We want OpenWebUI to send traces (conversations, latencies, costs, feedback) to Langfuse:

* **Langfuse Deployment:** We have Langfuse running (architecture shows it at port 3000). Langfuse requires a Postgres and ClickHouse (the architecture includes clickhouse and probably a langfuse service with those). Assuming that’s set up and Langfuse is reachable at e.g. http://langfuse:3000 (and you have API keys from its config).
* **OpenWebUI Integration via Pipelines:** OpenWebUI doesn’t send data to Langfuse by default; it needs the **Langfuse pipeline filter** installed. The OpenWebUI docs provide a guide[[66]](https://docs.openwebui.com/tutorials/integrations/langfuse/#:~:text=How%20to%20integrate%20Langfuse%20with,Open%20WebUI)[[45]](https://docs.openwebui.com/tutorials/integrations/langfuse/#:~:text=In%20the%20Admin%20Settings%2C%20create,API%20with%20the%20following%20details):
* First, run the **OpenWebUI Pipelines** container: ghcr.io/open-webui/pipelines:main on port 9099, as mentioned in the Langfuse tutorial[[67]](https://docs.openwebui.com/tutorials/integrations/langfuse/#:~:text=Launch%20Pipelines%20by%20using%20Docker,following%20command%20to%20start%20Pipelines). This container allows hooking custom Python code into the request/response flow.
* Connect OpenWebUI to that Pipelines server by adding a new model connection of type “OpenAI” in Admin settings, pointing to http://pipelines:9099 with the standard password (0p3n-w3bu!)[[45]](https://docs.openwebui.com/tutorials/integrations/langfuse/#:~:text=In%20the%20Admin%20Settings%2C%20create,API%20with%20the%20following%20details). This essentially chains requests through the pipeline server.
* In Admin -> Pipelines, add the **Langfuse Filter** from the examples repo[[46]](https://docs.openwebui.com/tutorials/integrations/langfuse/#:~:text=Step%204%3A%20Adding%20the%20Langfuse,Filter%20Pipeline). The URL provided[[68]](https://docs.openwebui.com/tutorials/integrations/langfuse/#:~:text=http%3A%2F%2Fhost,option%20with%20the%20following%20URL) points to a Python script (langfuse\_v3\_filter\_pipeline.py). When installed, you configure it with your Langfuse PUBLIC\_KEY and SECRET\_KEY (from Langfuse UI) within OpenWebUI.
* This filter will now capture each prompt/response and send trace data to your Langfuse server in real-time[[69]](https://docs.openwebui.com/tutorials/integrations/langfuse/#:~:text=Langfuse%20,use%20of%20Open%20WebUI%2C%20including). It can log token usage, response time, model details, etc., and even allow replaying conversations in the Langfuse dashboard.
* **Dashboard:** Once integrated, you can open Langfuse UI (likely at http://your-langfuse-domain) and see traces streaming in as you or users interact with OpenWebUI[[70]](https://docs.openwebui.com/tutorials/integrations/langfuse/#:~:text=,Evaluations). This gives insight into how often certain models are used, how long requests take, where errors occur, etc.
* **Feedback hooks:** Langfuse also can record ratings or feedback if you set that up (OpenWebUI has thumbs-up/down feedback on answers; those could be captured too, possibly via a custom action or pipeline).
* **Cost Monitoring:** Since OpenWebUI now uses pipelines, you can also utilize the **Usage** tracking. The guide mentions enabling a usage box in model settings to capture token counts[[71]](https://docs.openwebui.com/tutorials/integrations/langfuse/#:~:text=Image%3A%20Open%20WebUI%20add%20Langfuse,Pipeline). Langfuse will then show cost per conversation/user if configured with your pricing.
* **Other Observability:** We could also integrate **Helicone** (another monitoring tool) similarly, but since we chose Langfuse and have it, we proceed with that. In fact, Helicone integration is also possible (OpenWebUI docs mention it)[[72]](https://docs.openwebui.com/category/-integrations/#:~:text=use%20case,GitHub%29%20offers), but Langfuse being self-hosted aligns with our out-of-the-box goal.

By setting up Langfuse, our deployment is production-ready with observability. You can identify performance issues, track API usage (critical when using paid models via OpenRouter), and have an audit log of AI actions. This addresses tracing and debugging needs comprehensively[[73]](https://docs.openwebui.com/tutorials/integrations/langfuse/#:~:text=Langfuse%20,use%20of%20Open%20WebUI%2C%20including).

## Rate Limits & Cost Guardrails for Hosted LLMs

When using hosted models (OpenAI via OpenRouter, etc.), it’s wise to have **rate limiting and cost controls** to prevent abuse or runaway costs. We implement this through OpenWebUI’s pipeline filters or external tools:

* **Built-in Rate Limiting:** OpenWebUI’s pipelines include a **RateLimit filter** example[[74]](https://docs.openwebui.com/pipelines/#:~:text=WebUI%20docs,flow%20of%20requests%20to). You can install a filter that limits the number of requests per user per minute, etc. For instance, allow only 5 GPT-4 calls per minute per user to avoid spam. This would involve adding the rate limit filter via the Pipelines UI (similar to Langfuse). The filter code can be found in the OpenWebUI examples (likely named ratelimit\_filter.py). Configure it with desired rules (the example probably uses an in-memory counter or Redis).
* **OpenAI/Router Side Limits:** OpenRouter doesn’t inherently limit per user (aside from your account limits). So control needs to be on our side. RateLimit filter will handle frequency. For cost, a filter could sum tokens and decide if a user’s monthly quota is exceeded – that would require storing usage per user. This could be done by writing a custom pipeline that accumulates costs (perhaps storing in a file or database, or using Langfuse’s data).
* **Helicone:** Another method is to route OpenAI calls through Helicone (a proxy that monitors and can enforce quotas). However, adding Helicone in between might complicate our setup (since we already go through OpenRouter). If needed, one could have OpenRouter route to a Helicone endpoint for OpenAI, but that’s too convoluted. Instead, we rely on our own tracking.
* **Manual Observation:** With Langfuse, you at least can *see* how much each conversation costs[[70]](https://docs.openwebui.com/tutorials/integrations/langfuse/#:~:text=,Evaluations). It doesn’t automatically stop anything, but you can be aware.
* **Enforcing Limits:** To truly enforce, you might script something: e.g., daily job checks Langfuse data or internal logs and if a user exceeded $X spend, adjust their role or disable their access. OpenWebUI does allow roles, so you could switch a heavy user to a role that has no model permissions (there is concept of roles and perhaps you can restrict certain models to admin role only, etc., though not sure if granular cost-based restriction exists).
* **Timeouts:** For runaway requests (like a user asks the code interpreter to do something that hangs), OpenWebUI has some built-in timeouts. You can configure CODE\_EXECUTION\_JUPYTER\_TIMEOUT (we set 60 seconds in example)[[48]](https://docs.openwebui.com/tutorials/jupyter/#:~:text=Open,jupyter). Also OPENAI\_TIMEOUT or similar environment for API calls can be set (to avoid waiting forever on model response). Ensure these are set reasonably (e.g., 30-60s for model calls).
* **Testing Limits:** Simulate a scenario: fire multiple queries quickly and see if the RateLimit filter triggers (it should refuse or delay requests, possibly the assistant will say “Too many requests, slow down”). Also test a long-running code – it should stop around the timeout.

By implementing these guardrails, we prevent both **excessive usage** (which could rack up cloud API bills) and **abuse** (which could degrade service for others). This is especially important if you open the system to multiple users. As an admin, you can relax these for yourself while keeping regular users in check. The system’s design allows flexible filters, so you can tune to whatever thresholds make sense.

*(No direct doc citation for the filter code itself, but the presence of rate limit features was noted in community discussions*[*[74]*](https://docs.openwebui.com/pipelines/#:~:text=WebUI%20docs,flow%20of%20requests%20to)*.)*

## Healthcheck and Reliability

To keep the system running smoothly, we ensure proper health checks and restart policies:

* **Health Check Endpoint:** OpenWebUI provides a health endpoint at /healthz which returns 200 OK if the service is up[[75]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=Load%20Balancer%20Config%3A). We will configure Docker/Traefik to use this. For example, in the docker-compose, add:
* healthcheck:  
   test: ["CMD", "curl", "-f", "http://localhost:8080/healthz"]   
   interval: 1m30s  
   timeout: 10s  
   retries: 3
* This will let Docker mark the container unhealthy if something is wrong, and you could auto-restart it.
* **Traefik Monitoring:** Traefik is already checking if the service responds (via the routing rules). We can also use Traefik’s /api/version or similar to ensure Traefik itself is healthy. Traefik provides metrics and a dashboard which we have on port 8080 (for internal checking).
* **Watchdog Scripts:** The user provided a checkTraefik.sh script[[76]](file://file-Qx3UVhAYoXYGUi2vjDY9Z5#:~:text=echo%20%22%22%20echo%20%22,w%20%22%25%7Bhttp_code%7D%22%20http%3A%2F%2F%24DOMAIN)[[77]](file://file-Qx3UVhAYoXYGUi2vjDY9Z5#:~:text=echo%20%22%22%20echo%20%22,w%20%22%25%7Bhttp_code%7D%22%20http%3A%2F%2F%24DOMAIN%2Fapi) that pings the domain and /api path. This is a good practice – schedule that script via cron to run periodically. It verifies DNS, HTTP 200 on main page, 200 on /api/ path, port connectivity, etc. The output will quickly show if something’s off (e.g., API returning 502).
* **Restart Policies:** Ensure all Docker services have restart: unless-stopped (or always) so they come back on reboot or if they crash.
* **Load Balancer/WebSocket:** If in future you scale out OpenWebUI, note that sticky sessions aren’t needed due to the shared Redis approach (which we have set up). But for now, one instance is fine.
* **Backup/Updates:** Keep an eye on updates – OpenWebUI is active; occasionally update the image to get the latest fixes (but always backup config DB first). A quick health test after any update is to hit /healthz or even /api/ as done in the script to make sure all components (DB connections, etc.) initialize correctly.

With these, the environment should be **green on health** most of the time. You’ll get alerted by your checks if something fails.

*(Citing [43†L140-L147] for health check path confirmation.)*

## Backup and Restore Strategy

Given the complex setup, having a clear backup strategy is essential:

* **OpenWebUI Data:** If using the default volume for OpenWebUI (open-webui volume), it contains the SQLite database (if not external Postgres) and uploaded files (if not using S3). To backup: stop the container (to flush writes) or at least pause usage, then **tar/zip the volume**. You can do this by e.g., docker run --rm -v open-webui:/data -v $(pwd):/backup alpine tar czf /backup/openwebui\_data\_$(date +%F).tar.gz /data. Store that archive safely. This captures conversation history, configs, etc. (It’s small if you primarily store config and some chat logs.)
* **Database (Postgres):** If you used Postgres for OpenWebUI or Langfuse, back those up via SQL dump or use volume backup similarly. For Postgres, the typical pg\_dump or a cron job can dump to a file that gets backed up.
* **Vector DB (Qdrant):** Qdrant data is on its volume. Back that up if your knowledge store is critical (you can always re-index documents if you lose it, but backup is faster). Use the same approach: tar the volume or use Qdrant’s backup API if it has one.
* **Other Services:**
* **Nextcloud** (if any research files were stored there),
* **Jupyter notebooks** (the jupyter\_data volume should be backed up or use Nextcloud to sync notebooks),
* **Redis** (mostly ephemeral cache/session, not critical to backup unless you want session continuity),
* **Traefik config** (if you have dynamic config files or let’s encrypt certs, backup the acme.json if using LE, although Cloudflare DNS + wildcard likely handles TLS in this case).
* **Supabase** (if used) – that’s a whole stack with its own backup needs (Postgres, storage).
* **User Files and Projects:** If users saved projects or data in the environment (in mounted volumes), ensure those are covered. For instance, user uploaded PDFs – if using S3, the S3 itself should have its own redundancy; if using local storage, they are in the openwebui volume we already plan to backup.
* **Restore Process:** Document how to restore quickly: e.g., spin up new VM, deploy stack, then restore volumes from backup tarballs, bring up containers. Test this process occasionally.
* **Version Control Config:** Keep a copy of your docker-compose.yml and any config files (maybe the user’s architecture files) in source control or backed up. The architecture is quite complex, so having Infrastructure-as-Code is beneficial. That way if the server dies, you can reconstruct the services with configuration from code.

Regular backups (perhaps weekly for databases, daily for critical user data) will ensure you can recover from failures without losing much. Also, before any major upgrade to OpenWebUI or related components, take a fresh backup.

*(No citations; this is general good practice given the environment.)*

## Reverse Proxy & Networking (TLS, Upload Size)

The Traefik reverse proxy is central to our deployment, handling TLS termination and routing. We need to ensure it’s configured to support large uploads and secure connections:

* **TLS/Certificates:** Traefik is already set to use a wildcard certificate via Let’s Encrypt (as gleaned from the Cloudflare setup). It’s working if your domain is serving over HTTPS. Check that all subdomains (open-webui, n8n, etc.) have proper DNS records (the wildcard covers them) and Traefik routers have tls.certresolver=myresolver labels (we saw for ortools service as example)[[78]](file://file-WMzDZFcgL5wfGRyF9vvLfA#:~:text=labels%3A%20,%22traefik.http.routers.ortools.tls.certresolver%3Dmyresolver). In short, confirm that when you access each service, it’s using a valid cert. This was probably done as part of initial setup (Cloudflare + DNS + Traefik).
* **Request Size Limits:** By default, Traefik v2 (and v3) do not buffer large requests unless configured. Large file uploads (over 100MB, etc.) can result in 413 errors at Traefik if not handled[[79]](https://doc.traefik.io/traefik/reference/routing-configuration/http/middlewares/buffering/#:~:text=If%20the%20request%20exceeds%20the,option). We need to enable the **Buffering middleware** in Traefik with high limits:
* Add a middleware in Traefik config (either static or via labels) for buffering. For example, define:
* labels:  
   - "traefik.http.middlewares.upload-buffer.buffering.maxRequestBodyBytes=0"  
   - "traefik.http.middlewares.upload-buffer.buffering.memRequestBodyBytes=104857600"
* Setting maxRequestBodyBytes to 0 means no limit (only limited by system)[[79]](https://doc.traefik.io/traefik/reference/routing-configuration/http/middlewares/buffering/#:~:text=If%20the%20request%20exceeds%20the,option). We allocate some memory threshold (100MB in this example) before buffering to disk.
* Then attach this middleware to the OpenWebUI router: e.g., traefik.http.routers.openwebui.middlewares=upload-buffer@docker. This will apply the rule to that service.
* Similarly, if Nextcloud or other service needs large uploads, do likewise for them.
* Note: Cloudflare’s proxy (if you’re proxying through Cloudflare) has its own upload size limits (100 MB on free, 500 MB on business, etc.)[[80]](https://github.com/traefik/traefik/issues/8005#:~:text=413%20Request%20Entity%20Too%20Large,contact%20Customer). If you plan to upload very large files (say 1 GB), and you’re on Cloudflare free, you will hit their cap. Solutions: either don’t proxy those subdomains through Cloudflare (set them to DNS-only in Cloudflare, so Traefik is hit directly), or upgrade plan. For mostly text and moderate PDF usage, staying under 100MB is fine (e.g., chunk PDFs).
* **WebSockets:** Ensure Traefik is configured to allow WebSocket upgrade (the ALB info in the Medium article suggests it's needed for real-time features like token streaming)[[75]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=Load%20Balancer%20Config%3A). Traefik by default will pass Upgrade headers; just ensure no conflicting middleware strips them. The checkTraefik script also verifies WebSocket by connecting to port 5000/5001 (maybe those are for dev testing).
* **Compression:** If bandwidth is a concern, Traefik can compress responses. But for binary data (images, audio) it’s not needed. For text responses compression is nice. Traefik has a Compress middleware that could be enabled globally.
* **CORS and Security Headers:** If any clients call the OpenWebUI APIs from JS frontends, configure CORS headers via Traefik Headers middleware as needed. Not a big issue if mainly using the web UI itself.
* **Traefik Dashboard Security:** If Traefik’s dashboard is enabled on :8080, consider restricting it (IP whitelist or basic auth) since it can show info about your services. In a closed environment it’s fine.

By fine-tuning Traefik, we ensure smooth networking: **HTTPS** for all services, ability to **upload large files** without errors[[79]](https://doc.traefik.io/traefik/reference/routing-configuration/http/middlewares/buffering/#:~:text=If%20the%20request%20exceeds%20the,option), and stable connections for streaming responses and voice (if using websockets for mic, etc., they go through WSS properly).

## Quick Launch from Command Palette (for Services/Actions)

Beyond the earlier “New Research Project” item, the idea of **Quick-launch from the Command Palette** can be generalized. We want common actions or service links accessible quickly in the UI for convenience:

* **Launching External Apps:** Suppose you often need to jump to Nextcloud, JupyterHub, or RStudio from the chat interface. While OpenWebUI doesn’t have built-in shortcuts to those, you could add them as **custom HTML or script**. A non-intrusive way: in the OpenWebUI “Tips & Tricks” or any static markdown, include hyperlinks to those services for the user. Not elegant, but functional.
* **OpenWebUI Palette Extensibility:** Currently, the command palette (Ctrl+K) likely has entries for “New Chat”, “Go to Settings”, etc. If the code is accessible, one could extend it to include custom commands (like “Open JupyterHub” that triggers a new tab to that URL). This would require modifying the frontend source and rebuilding, which might not be desired if sticking to standard distributions.
* **Alternative – Agent Approach:** You could ask the assistant itself to provide links when asked. For example, if you type “open jupyter”, the assistant can respond with the URL as a markdown link if it knows it. We can simply tell the assistant (via system prompt or a custom function) the addresses of these services. Then at least a user can query it. Not as slick as a palette, but workable.
* **n8n quick actions:** Possibly, what the user wants is to quickly trigger workflows (like we covered with /newproject). If there are multiple such actions, consider implementing each as a slash command or a small “function” the assistant can call. For instance, define a dummy OpenAPI tool “launcher” with endpoints like /launch?target=jupyter which returns “Launched Jupyter” and have the function marked as causing a side-effect (opening a URL might not be doable from server side, but you could return a link or send a webhook to a local system to open something).
* **User Training:** Document to your users (or yourself) what quick commands exist. E.g., /jupyter will yield the Jupyter URL, /nextcloud opens that, etc.

Given the system orientation, it might be acceptable to handle quick launching outside of OpenWebUI (like using browser bookmarks or an app portal). However, it’s a neat idea to have everything accessible via the chat interface. This remains a bit custom – weigh the effort vs benefit.

*(No citations, as this is again a customization beyond standard features.)*

## Tool Permissions and Security Hardening

With so many powerful tools integrated (code execution, filesystem access, web access, etc.), it’s crucial to enforce permissions and sandboxing:

* **Role-Based Access:** OpenWebUI supports user roles (pending, user, admin)[[8]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=). By default, any logged-in user might have access to all tools we added (especially if we made them global tools). We should restrict dangerous tools to admins. For instance, you might not want regular users executing arbitrary Python code or reading server files.
* In Admin panel, see if there’s a way to limit a tool to certain roles. If not built-in, an alternative is to run separate OpenWebUI instances: one user-facing with safer config (no direct FS or system tools), and one admin-facing with all tools. But that’s heavier.
* Another approach: modify the tools themselves to enforce rules (e.g., the filesystem tool could be configured to only allow read access in a certain directory, and not allow writes or listing sensitive paths).
* **Sandboxing Code:** Our Jupyter integration runs code in a container separate from OpenWebUI, which is good (so if someone does !rm -rf /, they’re in the Jupyter container, not the host; and that container is presumably unprivileged). Keep it that way: don’t mount sensitive host directories into Jupyter container except a specific work folder.
* **Network Egress:** Be mindful that if the assistant can execute code or HTTP calls, it could theoretically reach out to the internet or internal services. If this is an issue, consider network policies (Docker doesn’t easily restrict container egress without extra config). At least the OpenWebUI container itself is on a Docker network behind Traefik, but it can still initiate outbound connections. Using internal services is fine (that’s our design), but we wouldn’t want it to call, say, cloud APIs unexpectedly unless we allow it.
* **Updates and Patches:** Keep OpenWebUI updated for security patches. Also update underlying images (Jupyter, etc.) regularly.
* **Logging and Auditing:** With Langfuse, you have logs of all actions. That helps detect misuse. You might also log OpenWebUI system messages or errors to file for further auditing.
* **Disable Unneeded Features:** If there are OpenWebUI features not used, disable them. For instance, if you are not using the built-in browser-based code interpreter (they had an earlier version running in a sandbox, not needed since we use Jupyter), turn it off. Similarly, if not using certain model providers, no need to keep their API keys configured.
* **User Isolation:** If you did allow multiple users, remember that code execution and tools are shared resources. One user could interfere with another (e.g., modifying files). A full multi-tenant isolation is complex here – likely this is for a small team or single user scenario, which mitigates a lot of risk (you trust yourself/admin with the tools, and there aren’t random users to worry about).
* **Admin Interface:** Protect admin panel access. Only admin role can access it anyway; ensure you don't accidentally share admin credentials.

By carefully selecting which tools to expose and to whom, and sandboxing their capabilities, we maintain a balance: **maximal functionality** for those who need it, with **minimal risk**. This is an area where ongoing vigilance is required – as you add new tools, always consider the worst-case if a prompt is misused or a user tries something sneaky, and adjust accordingly.

**Conclusion:** We have assembled an **OpenWebUI-centered environment** with a rich feature set: persistent data storage, secure user management, both local and remote LLMs, advanced retrieval with vector search, live web access, image generation/editing, voice I/O, coding capabilities, workflow automation, graph databases, and monitoring – all integrated into a cohesive system. The references provided confirm the feasibility of each integration and offer guidance on the configurations used. This “AI research assistant” stack is ready to handle a wide range of tasks out of the box, with full control in your hands since it’s self-hosted.

[[1]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=settings%20panel%20to%20Ollama%20or,OpenAI%20compatible%20endpoints) [[4]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=4,a%20storage%20backend%20like%20S3) [[5]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=%60REDIS_SENTINEL_HOSTS%3Dredis,links%20%26%20search) [[6]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=SQL%20%28Postgres%29%20,links%20%26%20search) [[18]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=RAG%20Content%20Extraction%20Engine%3A%20You,mistral_ocr) [[20]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=High%20QPS%20vector%20search%3A%20The,pinecone) [[23]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=Embedding%20Model%20Engine%3A%20You%20don%E2%80%99t,openai) [[24]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=Task%20Model%20Configuration%3A%20Open%20WebUI,your%20use%20cases%20are%20met) [[26]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=,vector%20store%20to%20leverage%20reranking) [[33]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=Open%20WebUI%20is%20one%20of%2C,ecosystem%20of%20tools%20and%20functions) [[75]](https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced#:~:text=Load%20Balancer%20Config%3A) The SRE’s Guide to High Availability Open WebUI Deployment Architecture | by Taylor Wilsdon | Medium

<https://taylorwilsdon.medium.com/the-sres-guide-to-high-availability-open-webui-deployment-architecture-2ee42654eced>

[[2]](https://docs.openwebui.com/tutorials/s3-storage/#:~:text=which%20means%20Open,storage) [[3]](https://docs.openwebui.com/tutorials/s3-storage/#:~:text=To%20use%20Amazon%20S3%2C%20we,S3_BUCKET_NAME) Switching to S3 Storage | Open WebUI

<https://docs.openwebui.com/tutorials/s3-storage/>

[[7]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=) [[8]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=) [[9]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=) [[10]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=) [[11]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=) [[12]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=match%20at%20L538%20,Else%20%60http%3A%2F%2Fhost.docker.internal%3A11434) [[13]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=and%20Ollama%20API%20endpoints%20,fresh%20API%20calls%20each%20time) [[14]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=) [[15]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=) [[16]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=) [[17]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=) [[19]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=,variable) [[21]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=) [[22]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=) [[25]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=%2A%20%60ollama%60%20,variable) [[27]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=) [[28]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=) [[29]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=) [[30]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=) [[31]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=) [[32]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=) [[34]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=%2A%20%60comfyui%60%20,Default%3A%20%60openai) [[35]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=) [[36]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=) [[52]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=) [[53]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=) [[54]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=,variable) [[55]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=,variable) [[56]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=%2A%20%60elevenlabs%60%20,Speech%20engine%20to%20use) [[60]](https://docs.openwebui.com/getting-started/env-configuration/#vector-database#:~:text=) Environment Variable Configuration | Open WebUI

<https://docs.openwebui.com/getting-started/env-configuration/>

[[37]](https://docs.openwebui.com/openapi-servers/mcp/#:~:text=This%20documentation%20explains%20how%20to,users%20and%20developers) [[39]](https://docs.openwebui.com/openapi-servers/mcp/#:~:text=uvx%20mcpo%20,your_mcp_server_command) [[40]](https://docs.openwebui.com/openapi-servers/mcp/#:~:text=You%20can%20easily%20run%20the,OpenAPI%20proxy%20%28%60mcpo%60%29%20like%20this) [[43]](https://docs.openwebui.com/openapi-servers/mcp/#:~:text=,specific%20glue%20code) [[44]](https://docs.openwebui.com/openapi-servers/mcp/#:~:text=,Happier%20developers%20%26%20users%20%E2%9C%85) ️ MCP Support | Open WebUI

<https://docs.openwebui.com/openapi-servers/mcp/>

[[38]](https://docs.openwebui.com/openapi-servers/open-webui/#:~:text=Open%20WebUI%20v0,powered%20tool%20servers) [[41]](https://docs.openwebui.com/openapi-servers/open-webui/#:~:text=Image%3A%20Settings%20Page) [[42]](https://docs.openwebui.com/openapi-servers/open-webui/#:~:text=) Open WebUI Integration | Open WebUI

<https://docs.openwebui.com/openapi-servers/open-webui/>

[[45]](https://docs.openwebui.com/tutorials/integrations/langfuse/#:~:text=In%20the%20Admin%20Settings%2C%20create,API%20with%20the%20following%20details) [[46]](https://docs.openwebui.com/tutorials/integrations/langfuse/#:~:text=Step%204%3A%20Adding%20the%20Langfuse,Filter%20Pipeline) [[66]](https://docs.openwebui.com/tutorials/integrations/langfuse/#:~:text=How%20to%20integrate%20Langfuse%20with,Open%20WebUI) [[67]](https://docs.openwebui.com/tutorials/integrations/langfuse/#:~:text=Launch%20Pipelines%20by%20using%20Docker,following%20command%20to%20start%20Pipelines) [[68]](https://docs.openwebui.com/tutorials/integrations/langfuse/#:~:text=http%3A%2F%2Fhost,option%20with%20the%20following%20URL) [[69]](https://docs.openwebui.com/tutorials/integrations/langfuse/#:~:text=Langfuse%20,use%20of%20Open%20WebUI%2C%20including) [[70]](https://docs.openwebui.com/tutorials/integrations/langfuse/#:~:text=,Evaluations) [[71]](https://docs.openwebui.com/tutorials/integrations/langfuse/#:~:text=Image%3A%20Open%20WebUI%20add%20Langfuse,Pipeline) [[73]](https://docs.openwebui.com/tutorials/integrations/langfuse/#:~:text=Langfuse%20,use%20of%20Open%20WebUI%2C%20including) Monitoring and Debugging with Langfuse | Open WebUI

<https://docs.openwebui.com/tutorials/integrations/langfuse/>

[[47]](https://docs.openwebui.com/tutorials/jupyter/#:~:text=jupyter%3A%20image%3A%20jupyter%2Fminimal,JUPYTER_TOKEN%3D123456) [[48]](https://docs.openwebui.com/tutorials/jupyter/#:~:text=Open,jupyter) [[49]](https://docs.openwebui.com/tutorials/jupyter/#:~:text=Jupyter%20Notebook%20Integration) [[50]](https://docs.openwebui.com/tutorials/jupyter/#:~:text=jupyter%3A%20image%3A%20jupyter%2Fminimal,JUPYTER_ENABLE_LAB%3Dyes) [[51]](https://docs.openwebui.com/tutorials/jupyter/#:~:text=Jupyter%20Notebook%20is%20an%20open,thought%20process%20and%20findings%20along) Jupyter Notebook Integration | Open WebUI

<https://docs.openwebui.com/tutorials/jupyter/>

[[57]](https://docs.openwebui.com/tutorials/text-to-speech/openedai-speech-integration/#:~:text=info) [[58]](https://docs.openwebui.com/tutorials/text-to-speech/openedai-speech-integration/#:~:text=You%20can%20use%20any%20of,the%20following%20Docker%20Compose%20files) [[59]](https://docs.openwebui.com/tutorials/text-to-speech/openedai-speech-integration/#:~:text=) [[61]](https://docs.openwebui.com/tutorials/text-to-speech/openedai-speech-integration/#:~:text=match%20at%20L249%20,TTS%20also%20supports%20multilingual%20voices) [[62]](https://docs.openwebui.com/tutorials/text-to-speech/openedai-speech-integration/#:~:text=,speech%60%20service) [[65]](https://docs.openwebui.com/tutorials/text-to-speech/openedai-speech-integration/#:~:text=,1GB%20vs.%208GB) ️ Openedai-speech Using Docker | Open WebUI

<https://docs.openwebui.com/tutorials/text-to-speech/openedai-speech-integration/>

[[63]](https://github.com/open-webui/open-webui/discussions/12937#:~:text=Introducing%20Custom%20TTS%20Engine%20Support%21,Custom%20TTS%20API%20Key%3A) Introducing Custom TTS Engine Support! (OpenAPI Compatible)

<https://github.com/open-webui/open-webui/discussions/12937>

[[64]](https://www.reddit.com/r/LocalLLaMA/comments/1dfa02r/openaicompatible_tts_wrapper_ie_for_openwebuis/#:~:text=feature%29%20www,TTS%20Settings%20and%20connect%20it) OpenAI-compatible TTS wrapper (i.e. for OpenWebUIs call feature)

<https://www.reddit.com/r/LocalLLaMA/comments/1dfa02r/openaicompatible_tts_wrapper_ie_for_openwebuis/>

[[72]](https://docs.openwebui.com/category/-integrations/#:~:text=use%20case,GitHub%29%20offers) Integrations | Open WebUI

<https://docs.openwebui.com/category/-integrations/>

[[74]](https://docs.openwebui.com/pipelines/#:~:text=WebUI%20docs,flow%20of%20requests%20to) Pipelines: UI-Agnostic OpenAI API Plugin Framework - Open WebUI

<https://docs.openwebui.com/pipelines/>

[[76]](file://file-Qx3UVhAYoXYGUi2vjDY9Z5#:~:text=echo%20%22%22%20echo%20%22,w%20%22%25%7Bhttp_code%7D%22%20http%3A%2F%2F%24DOMAIN) [[77]](file://file-Qx3UVhAYoXYGUi2vjDY9Z5#:~:text=echo%20%22%22%20echo%20%22,w%20%22%25%7Bhttp_code%7D%22%20http%3A%2F%2F%24DOMAIN%2Fapi) checkTraefic.sh

<file://file-Qx3UVhAYoXYGUi2vjDY9Z5>

[[78]](file://file-WMzDZFcgL5wfGRyF9vvLfA#:~:text=labels%3A%20,%22traefik.http.routers.ortools.tls.certresolver%3Dmyresolver) docker-compose.yml

<file://file-WMzDZFcgL5wfGRyF9vvLfA>

[[79]](https://doc.traefik.io/traefik/reference/routing-configuration/http/middlewares/buffering/#:~:text=If%20the%20request%20exceeds%20the,option) Traefik Buffering Documentation - Traefik

<https://doc.traefik.io/traefik/reference/routing-configuration/http/middlewares/buffering/>

[[80]](https://github.com/traefik/traefik/issues/8005#:~:text=413%20Request%20Entity%20Too%20Large,contact%20Customer) 413 Request Entity Too Large · Issue #8005 · traefik/traefik - GitHub

<https://github.com/traefik/traefik/issues/8005>