What is OpenRAG?

OpenRAG is an open-source package for building agentic RAG systems that integrates with a wide range of orchestration tools, vector databases, and LLM providers.

OpenRAG connects and amplifies three popular, proven open-source projects into one powerful platform:

- Langflow: Langflow is a versatile tool for building and deploying AI agents and MCP servers. It supports all major LLMs, vector databases, and a growing library of AI tools.

  OpenRAG uses several built-in flows, and it provides full access to all Langflow features through the embedded Langflow visual editor.

  By customizing the built-in flows or creating your own flows, every part of the OpenRAG stack interchangeable. You can modify any aspect of the flows from basic settings, like changing the language model, to replacing entire components. You can also write your own custom Langflow components, integrate MCP servers, call APIs, and leverage any other functionality provided by Langflow.

- OpenSearch: OpenSearch is a community-driven, Apache 2.0-licensed open source search and analytics suite that makes it easy to ingest, search, visualize, and analyze data. It provides powerful hybrid search capabilities with enterprise-grade security and multi-tenancy support.

  OpenRAG uses OpenSearch as the underlying vector database for storing and retrieving your documents and associated vector data (embeddings). You can ingest documents from a variety of sources, including your local filesystem and OAuth authenticated connectors to popular cloud storage services.

- Docling: Docling simplifies document processing, supports many file formats and advanced PDF parsing, and provides seamless integrations with the generative AI ecosystem.

  OpenRAG uses Docling to parse and chunk documents that are stored in your OpenSearch knowledge base.

> **TIP**
>
> Ready to get started? Try the quickstart to install OpenRAG and start exploring in minutes.

OpenRAG architecture

OpenRAG deploys and orchestrates a lightweight, container-based architecture that combines **Langflow**, **OpenSearch**, and **Docling** into a cohesive RAG platform.

- **OpenRAG backend**: The central orchestration service that coordinates all other components.
- **Langflow**: This container runs a Langflow instance. It provides the embedded Langflow visual editor for editing and creating flow, and it connects to the **OpenSearch** container for vector storage and retrieval.
- **Docling Serve**: This is a local document processing service managed by the **OpenRAG backend**.
- **External connectors**: Integrate third-party cloud storage services with OAuth authenticated connectors to the **OpenRAG backend**, allowing you to load documents from external storage to your OpenSearch knowledge base.
- **OpenRAG frontend**: Provides the user interface for interacting with the OpenRAG platform.

Quickstart

Use this quickstart to install OpenRAG, and then try some of OpenRAG's core features.

Prerequisites

This quickstart requires the following:

- An OpenAI API key. This quickstart uses OpenAI for simplicity. For other providers, see the complete installation guide.
- Python version 3.13 or later.
- Microsoft Windows only: To run OpenRAG on Windows, you must use the Windows Subsystem for Linux (WSL).

Install WSL for OpenRAG

i. Install WSL with the Ubuntu distribution using WSL 2:

```
wsl --install -d Ubuntu
```

For new installations, the `wsl --install` command uses WSL 2 and Ubuntu by default.

For existing WSL installations, you can change the distribution and check the WSL version.

> **KNOWN LIMITATION**
> OpenRAG isn't compatible with nested virtualization, which can cause networking issues. Don't install OpenRAG on a WSL distribution that is installed inside a Windows VM. Instead, install OpenRAG on your base OS or a non-nested Linux VM.

ii. Start your WSL Ubuntu distribution if it doesn't start automatically.

iii. Set up a username and password for your WSL distribution.

iv. Install Docker Desktop for Windows with WSL 2. When you reach the Docker Desktop **WSL integration** settings, make sure your Ubuntu distribution is enabled, and then click **Apply & Restart** to enable Docker support in WSL.

v. Install and run OpenRAG from within your WSL Ubuntu distribution.

If you encounter issues with port forwarding or the Windows Firewall, you might need to adjust the Hyper-V firewall settings to allow communication between your WSL distribution and the Windows host. For more troubleshooting advice for networking issues, see Troubleshooting WSL common issues.

Install OpenRAG

For this quickstart, install OpenRAG with the automatic installer script and basic setup:

1. Create a directory to store the OpenRAG configuration files, and then change to that directory:

```
mkdir openrag-workspace
cd openrag-workspace
```

2. Download the OpenRAG install script, move it to your OpenRAG directory, and then run it:

```
bash run_openrag_with_prereqs.sh
```

This script installs OpenRAG and its dependencies, including Docker or Podman, and it creates a `.env` file and `docker-compose` files in the current working directory. You might be prompted to install certain dependencies if they aren't already present in your environment. This process can take a few minutes. Once the environment is ready, OpenRAG starts.

3. Click **Basic Setup**.

4. Create passwords for your OpenRAG installation's OpenSearch and Langflow services. You can click **Generate Passwords** to automatically generate passwords.

The OpenSearch password is required. The Langflow admin password is optional. If you don't generate a Langflow admin password, Langflow runs in autologin mode with no password required.

Your passwords are saved in the `.env` file that is used to start OpenRAG. You can find this file in your OpenRAG installation directory.

5. Click **Save Configuration**, and then click **Start All Services**.

   Wait a few minutes while the startup process pulls and runs the necessary container images. Proceed when you see the following messages in the terminal user interface (TUI):

   ```
   Services started successfully
   Command completed successfully
   ```

6. To open the OpenRAG application, go to the TUI main menu, and then click **Open App**. Alternatively, in your browser, navigate to `localhost:3000`.

7. Select the **OpenAI** model provider, enter your OpenAI API key, and then click **Complete**.

   For this quickstart, you can use the default options for the model settings.

8. Click through the overview slides for a brief introduction to OpenRAG and basic setup, or click → **Skip overview**. You can complete this quickstart without going through the overview.

Load and chat with documents

Use the OpenRAG **Chat** to explore the documents in your OpenRAG database using natural language queries. Some documents are included by default to get you started, and you can load your own documents.

1. In OpenRAG, click **Chat**.

2. For this quickstart, ask the agent what documents are available. For example: `What documents are available to you?`

The agent responds with a summary of OpenRAG's default documents.

3. To verify the agent's response, click **Knowledge** to view the documents stored in the OpenRAG OpenSearch vector database. You can click a document to view the chunks of the document as they are stored in the database.

4. Click **Add Knowledge** to add your own documents to your OpenRAG knowledge base.

   For this quickstart, use either the **File** or **Folder** upload options to load documents from your local machine. **Folder** uploads an entire directory. The default directory is the `/openrag-documents` subdirectory in your OpenRAG installation directory.

   For information about the cloud storage provider options, see Ingest files with OAuth connectors.

5. Return to the **Chat** window, and then ask a question related to the documents that you just uploaded.

   If the agent's response doesn't seem to reference your documents correctly, try the following:

   - Click **Function Call: search_documents (tool_call)** to view the log of tool calls made by the agent. This is helpful for troubleshooting because it shows you how the agent used particular tools.

   - Click **Knowledge** to confirm that the documents are present in the OpenRAG OpenSearch vector database, and then click each document to see how the document was chunked. If a document was chunked improperly, you might need to tweak the ingestion or modify and reupload the document.

   - Click **Settings** to modify the knowledge ingestion settings.

   For more information, see Configure knowledge and Ingest knowledge.

Change the language model and chat settings

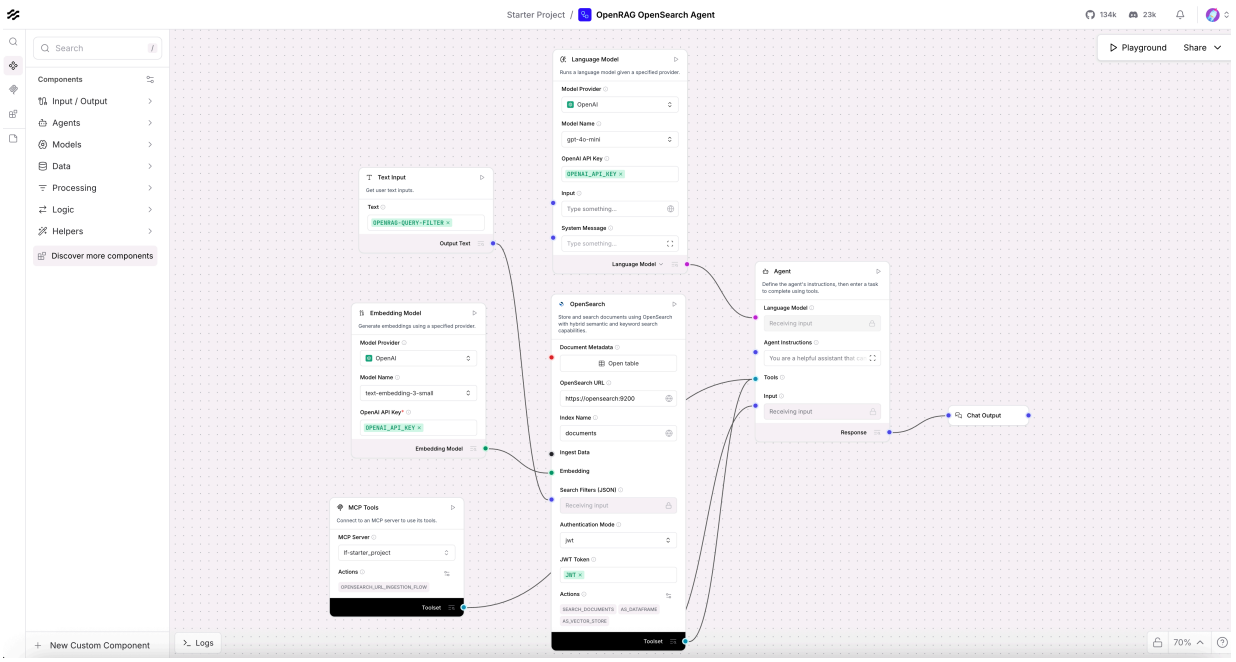
1. To change the knowledge ingestion settings, agent behavior, or language model, click **Settings**.

   The **Settings** page provides quick access to commonly used parameters like the **Language model** and **Agent Instructions**.

2. For greater insight into the underlying Langflow flow that drives the OpenRAG chat, click **Edit in Langflow** and then click **Proceed** to launch the Langflow visual editor in a new browser window.

   If Langflow requests login information, enter the `LANGFLOW_SUPERUSER` and `LANGFLOW_SUPERUSER_PASSWORD` from the `.env` file in your OpenRAG installation directory.

   The **OpenRAG OpenSearch Agent** flow opens in a new browser window.



3. For this quickstart, try changing the model. Click the **Language Model** component, and then change the **Model Name** to a different OpenAI model.

   After you edit a built-in flow, you can click **Restore flow** on the **Settings** page to revert the flow to its original state when you first installed OpenRAG.

4. Press `Command`+`S` (`Ctrl`+`S`) to save your changes.

You can close the Langflow browser window, or leave it open if you want to continue experimenting with the flow editor.

5. Switch to your OpenRAG browser window, and then click + in the **Conversations** tab to start a new conversation. This ensures that the chat doesn't persist any context from the previous conversation with the original model.

6. Ask the same question you asked in Load and chat with documents to see how the response differs from the original model.

Integrate OpenRAG into an application

Langflow in OpenRAG includes pre-built flows that you can integrate into your applications using the Langflow API. You can use these flows as-is or modify them to better suit your needs, as demonstrated in Change the language model and chat settings.

You can send and receive requests with the Langflow API using Python, TypeScript, or curl.

1. Open the **OpenRAG OpenSearch Agent** flow in the Langflow visual editor: From the **Chat** window, click **Settings**, click **Edit in Langflow**, and then click **Proceed**.

2. Create a Langflow API key, which is a user-specific token required to send requests to the Langflow server. This key doesn't grant access to OpenRAG.

    i. In the Langflow visual editor, click your user icon in the header, and then select **Settings**.
    ii. Click **Langflow API Keys**, and then click + **Add New**.
    iii. Name your key, and then click **Create API Key**.
    iv. Copy the API key and store it securely.
    v. Exit the Langflow **Settings** page to return to the visual editor.

3. Click **Share**, and then select **API access** to get pregenerated code snippets that call the Langflow API and run the flow.

    These code snippets construct API requests with your Langflow server URL (`LANGFLOW_SERVER_ADDRESS`), the flow to run (`FLOW_ID`), required headers (`LANGFLOW_API_KEY`, `Content-Type`), and a payload containing the required inputs to run the flow, including a default chat input message.

In production, you would modify the inputs to suit your application logic. For example, you could replace the default chat input message with dynamic user input.

Python

```
import requests
import os
import uuid
api_key = 'LANGFLOW_API_KEY'
url = "http://LANGFLOW_SERVER_ADDRESS/api/v1/run/FLOW_ID"  #
The complete API endpoint URL for this flow
# Request payload configuration
payload = {
    "output_type": "chat",
    "input_type": "chat",
    "input_value": "hello world!"
}
payload["session_id"] = str(uuid.uuid4())
headers = {"x-api-key": api_key}
try:
    # Send API request
    response = requests.request("POST", url, json=payload,
headers=headers)
    response.raise_for_status()  # Raise exception for bad
status codes
    # Print response
    print(response.text)
except requests.exceptions.RequestException as e:
    print(f"Error making API request: {e}")
except ValueError as e:
    print(f"Error parsing response: {e}")
```

TypeScript

```
const crypto = require('crypto');
const apiKey = 'LANGFLOW_API_KEY';
const payload = {
    "output_type": "chat",
    "input_type": "chat",
    "input_value": "hello world!"
};
payload.session_id = crypto.randomUUID();
```

```
const options = {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json',
        "x-api-key": apiKey
    },
    body: JSON.stringify(payload)
};
fetch('http://LANGFLOW_SERVER_ADDRESS/api/v1/run/FLOW_ID',
options)
    .then(response => response.json())
    .then(response => console.warn(response))
    .catch(err => console.error(err));
```

curl

```
curl --request POST \
  --url 'http://LANGFLOW_SERVER_ADDRESS/api/v1/run/FLOW_ID?
stream=false' \
  --header 'Content-Type: application/json' \
  --header "x-api-key: LANGFLOW_API_KEY" \
  --data '{
    "output_type": "chat",
    "input_type": "chat",
    "input_value": "hello world!"
  }'
```

4. Copy your preferred snippet, and then run it:

    - **Python**: Paste the snippet into a `.py` file, save it, and then run it with `python filename.py`.
    - **TypeScript**: Paste the snippet into a `.ts` file, save it, and then run it with `ts-node filename.ts`.
    - **curl**: Paste and run snippet directly in your terminal.

If the request is successful, the response includes many details about the flow run, including the session ID, inputs, outputs, components, durations, and more.

In production, you won't pass the raw response to the user in its entirety. Instead, you extract and reformat relevant fields for different use cases, as demonstrated in the Langflow quickstart. For example, you could pass the chat output text to a front-end

user-facing application, and store specific fields in logs and backend data stores for monitoring, chat history, or analytics. You could also pass the output from one flow as input to another flow.

Next steps

- **Reinstall OpenRAG with your preferred settings**: This quickstart used a minimal setup to demonstrate OpenRAG's core functionality. It is recommended that you reinstall OpenRAG with your preferred configuration because some settings are immutable after initial setup. For all installation options, see Install OpenRAG with TUI and Install OpenRAG with containers.

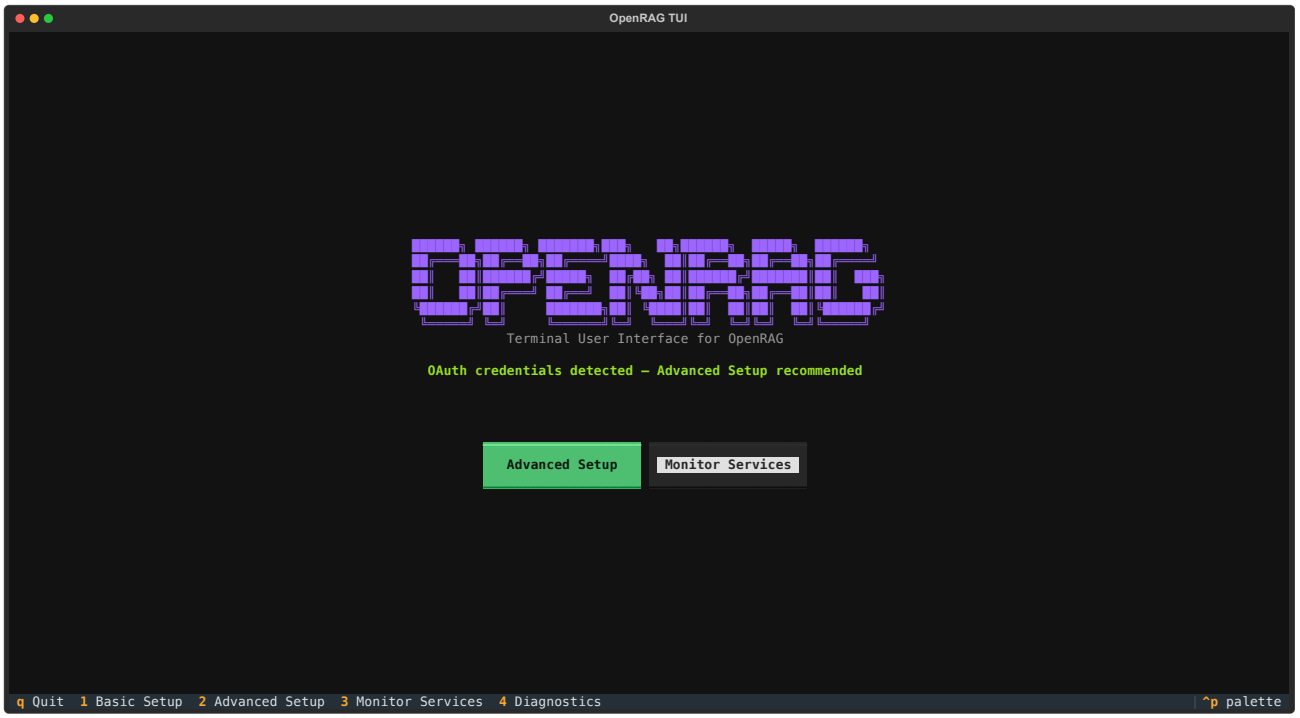
- **Learn more about OpenRAG**: Explore OpenRAG and the OpenRAG documentation to learn more about its features and functionality.

- **Learn more about Langflow**: For a deep dive on the Langflow API and visual editor, see the Langflow documentation.

Install OpenRAG with TUI

Install OpenRAG and then run the OpenRAG Terminal User Interface(TUI) to start your OpenRAG deployment with a guided setup process.

The OpenRAG Terminal User Interface (TUI) allows you to set up, configure, and monitor your OpenRAG deployment directly from the terminal.



Instead of starting OpenRAG using Docker commands and manually editing values in the `.env` file, the TUI walks you through the setup. It prompts for variables where required, creates a `.env` file for you, and then starts OpenRAG.

Once OpenRAG is running, use the TUI to monitor your application, control your containers, and retrieve logs.

If you prefer running Podman or Docker containers and manually editing `.env` files, see Install OpenRAG Containers.

Prerequisites

- All OpenRAG installations require Python version 3.13 or later.
- If you aren't using the automatic installer script, install the following:

    - uv.
    - Podman (recommended) or Docker.
    - `podman-compose` or Docker Compose. To use Docker Compose with Podman, you must alias Docker Compose commands to Podman commands.
- Microsoft Windows only: To run OpenRAG on Windows, you must use the Windows Subsystem for Linux (WSL).

Install WSL for OpenRAG

i. Install WSL with the Ubuntu distribution using WSL 2:

```
wsl --install -d Ubuntu
```

For new installations, the `wsl --install` command uses WSL 2 and Ubuntu by default.

For existing WSL installations, you can change the distribution and check the WSL version.

> **KNOWN LIMITATION**
> OpenRAG isn't compatible with nested virtualization, which can cause networking issues. Don't install OpenRAG on a WSL distribution that is installed inside a Windows VM. Instead, install OpenRAG on your base OS or a non-nested Linux VM.

ii. Start your WSL Ubuntu distribution if it doesn't start automatically.

iii. Set up a username and password for your WSL distribution.

iv. Install Docker Desktop for Windows with WSL 2. When you reach the Docker Desktop **WSL integration** settings, make sure your Ubuntu distribution is enabled, and then click **Apply & Restart** to enable Docker support in WSL.

v. Install and run OpenRAG from within your WSL Ubuntu distribution.

If you encounter issues with port forwarding or the Windows Firewall, you might need to adjust the Hyper-V firewall settings to allow communication between your WSL distribution and the Windows host. For more troubleshooting advice for networking issues, see Troubleshooting WSL common issues.

- Prepare model providers and credentials.

  During application onboarding, you must select language model and embedding model providers. If your chosen provider offers both types, you can use the same provider for both selections. If your provider offers only one type, such as Anthropic, you must select two providers.

  Gather the credentials and connection details for your chosen model providers before starting onboarding:

  - OpenAI: Create an OpenAI API key.
  - Anthropic language models: Create an Anthropic API key.
  - IBM watsonx.ai: Get your watsonx.ai API endpoint, IBM project ID, and IBM API key from your watsonx deployment.
  - Ollama: Use the Ollama documentation to set up your Ollama instance locally, in the cloud, or on a remote server, and then get your Ollama server's base URL.
- Optional: Install GPU support with an NVIDIA GPU, CUDA support, and compatible NVIDIA drivers on the OpenRAG host machine. If you don't have GPU capabilities, OpenRAG provides an alternate CPU-only deployment.

Install OpenRAG

Choose an installation method based on your needs:

- For new users, the automatic installer script detects and installs prerequisites and then runs OpenRAG.
- For a quick test, use `uvx` to run OpenRAG without creating a project or modifying files.
- Use `uv add` to install OpenRAG as a managed dependency in a new or existing Python project.
- Use `uv pip install` to install OpenRAG into an existing virtual environment.

Automatic installer

The script detects and installs uv, Docker/Podman, and Docker Compose prerequisites, then runs OpenRAG with `uvx`.

1. Create a directory to store the OpenRAG configuration files:

   ```
   mkdir openrag-workspace
   cd openrag-workspace
   ```

2. Run the installer:

   ```
   curl -fsSL
   https://docs.openr.ag/files/run_openrag_with_prereqs.sh | bash
   ```

The TUI creates a `.env` file and docker-compose files in the current working directory.

Quick test with uvx

Use `uvx` to quickly run OpenRAG without creating a project or modifying any files.

1. Create a directory to store the OpenRAG configuration files:

   ```
   mkdir openrag-workspace
   cd openrag-workspace
   ```

2. Run OpenRAG:

   ```
   uvx openrag
   ```

   To run a specific version:

   ```
   uvx --from openrag==0.1.30 openrag
   ```

The TUI creates a `.env` file and docker-compose files in the current working directory.

Python project with uv add

Use `uv add` to install OpenRAG as a dependency in your Python project. This adds OpenRAG to your `pyproject.toml` and lockfile, making your installation reproducible and version-controlled.

1. Create a new project with a virtual environment:

   ```
   uv init YOUR_PROJECT_NAME
   cd YOUR_PROJECT_NAME
   ```

   The `(venv)` prompt doesn't change, but `uv` commands will automatically use the project's virtual environment.

2. Add OpenRAG to your project:

   ```
   uv add openrag
   ```

   To add a specific version:

   ```
   uv add openrag==0.1.30
   ```

3. Start the OpenRAG TUI:

   ```
   uv run openrag
   ```

Install a local wheel

If you downloaded the OpenRAG wheel to your local machine, install it by specifying its path:

1. Add the wheel to your project:

   ```
   uv add PATH/TO/openrag-VERSION-py3-none-any.whl
   ```

   Replace `PATH/TO/` and `VERSION` with the path and version of your downloaded OpenRAG `.whl` file.

2. Run OpenRAG:

```
uv run openrag
```

Existing virtual environment with uv pip install

Use `uv pip install` to install OpenRAG into an existing virtual environment that isn't managed by `uv`.

> **TIP**
> For new projects, `uv add` is recommended as it manages dependencies in your project's lockfile.

1. Activate your virtual environment.

2. Install OpenRAG:

```
uv pip install openrag
```

3. Run OpenRAG:

```
uv run openrag
```

Continue with Set up OpenRAG with the TUI.

If you encounter errors during installation, see Troubleshoot OpenRAG.

Set up OpenRAG with the TUI

The OpenRAG setup process creates a `.env` file at the root of your OpenRAG directory, and then starts OpenRAG. If it detects a `.env` file in the OpenRAG root directory, it sources any variables from the `.env` file.

The TUI offers two setup methods to populate the required values. **Basic Setup** can generate all minimum required values for OpenRAG. However, **Basic Setup** doesn't enable OAuth connectors for cloud storage. If you want to use OAuth connectors to

upload documents from cloud storage, select **Advanced Setup**. If OpenRAG detects OAuth credentials, it recommends **Advanced Setup**.

Basic setup

1. To install OpenRAG with **Basic Setup**, click **Basic Setup** or press `1`.
2. Click **Generate Passwords** to generate passwords for OpenSearch and Langflow.

   The OpenSearch password is required. The Langflow admin password is optional. If no Langflow admin password is generated, Langflow runs in autologin mode with no password required.
3. Optional: Paste your OpenAI API key in the OpenAI API key field. You can also provide this during onboarding or choose a different model provider.
4. Click **Save Configuration**. Your passwords are saved in the `.env` file used to start OpenRAG.
5. To start OpenRAG, click **Start All Services**. Startup pulls container images and runs them, so it can take some time. When startup is complete, the TUI displays the following:

   ```
   Services started successfully
   Command completed successfully
   ```
6. To start the Docling service, under **Native Services**, click **Start**.
7. To open the OpenRAG application, navigate to the TUI main menu, and then click **Open App**. Alternatively, in your browser, navigate to `localhost:3000`.
8. Continue with application onboarding.

Advanced setup

1. To install OpenRAG with **Advanced Setup**, click **Advanced Setup** or press `2`.
2. Click **Generate Passwords** to generate passwords for OpenSearch and Langflow.

The OpenSearch password is required. The Langflow admin password is optional. If no Langflow admin password is generated, Langflow runs in autologin mode with no password required.

3. Paste your OpenAI API key in the OpenAI API key field.

4. If you want to upload documents from external storage, such as Google Drive, add the required OAuth credentials for the connectors that you want to use. These settings can be populated automatically if OpenRAG detects these credentials in a `.env` file in the OpenRAG installation directory.

    - **Amazon**: Provide your AWS Access Key ID and AWS Secret Access Key with access to your S3 instance. For more information, see the AWS documentation on Configuring access to AWS applications.
    - **Google**: Provide your Google OAuth Client ID and Google OAuth Client Secret. You can generate these in the Google Cloud Console. For more information, see the Google OAuth client documentation.
    - **Microsoft**: For the Microsoft OAuth Client ID and Microsoft OAuth Client Secret, provide Azure application registration credentials for SharePoint and OneDrive. For more information, see the Microsoft Graph OAuth client documentation.

    You can manage OAuth credentials later, but it is recommended to configure them during initial set up.

5. The OpenRAG TUI presents redirect URIs for your OAuth app. These are the URLs your OAuth provider will redirect back to after user sign-in. Register these redirect values with your OAuth provider as they are presented in the TUI.

6. Click **Save Configuration**.

7. To start OpenRAG, click **Start All Services**. Startup pulls container images and runs them, so it can take some time. When startup is complete, the TUI displays the following:

    ```
    Services started successfully
    Command completed successfully
    ```

8. To start the Docling service, under **Native Services**, click **Start**.

9. To open the OpenRAG application, navigate to the TUI main menu, and then click **Open App**. Alternatively, in your browser, navigate to `localhost:3000`.

10. If you enabled OAuth connectors, you must sign in to your OAuth provider before being redirected to your OpenRAG instance.

11. Two additional variables are available for **Advanced Setup** at this point. Only change these variables if you have a non-default network configuration for your deployment, such as using a reverse proxy or custom domain.

    - `LANGFLOW_PUBLIC_URL`: Sets the base address to access the Langflow web interface. This is where users interact with flows in a browser.

    - `WEBHOOK_BASE_URL`: Sets the base address of the OpenRAG OAuth connector endpoint. Supported webhook endpoints:

      - Amazon S3: Not applicable.
      - Google Drive: `/connectors/google_drive/webhook`
      - OneDrive: `/connectors/onedrive/webhook`
      - SharePoint: `/connectors/sharepoint/webhook`

12. Continue with application onboarding.

Application onboarding

The first time you start OpenRAG, regardless of how you installed it, you must complete application onboarding.

Some of these variables, such as the embedding models, can be changed seamlessly after onboarding. Others are immutable and require you to destroy and recreate the OpenRAG containers. For more information, see Environment variables.

You can use different providers for your language model and embedding model, such as Anthropic for the language model and OpenAI for the embeddings model. Additionally, you can set multiple embedding models.

You only need to complete onboarding for your preferred providers.

Anthropic

> **INFO**
>
> Anthropic doesn't provide embedding models. If you select Anthropic for your language model, you must select a different provider for embeddings.

1. Enable **Use environment Anthropic API key** to automatically use your key from the `.env` file. Alternatively, paste an Anthropic API key into the field.
2. Under **Advanced settings**, select your **Language Model**.
3. Click **Complete**.
4. In the second onboarding panel, select a provider for embeddings and select your **Embedding Model**.
5. To complete the onboarding tasks, click **What is OpenRAG**, and then click **Add a Document**. Alternatively, click → **Skip overview**.
6. Continue with the Quickstart.

OpenAI

1. Enable **Get API key from environment variable** to automatically enter your key from the TUI-generated `.env` file. Alternatively, paste an OpenAI API key into the field.
2. Under **Advanced settings**, select your **Language Model**.
3. Click **Complete**.
4. In the second onboarding panel, select a provider for embeddings and select your **Embedding Model**.
5. To complete the onboarding tasks, click **What is OpenRAG**, and then click **Add a Document**. Alternatively, click → **Skip overview**.
6. Continue with the Quickstart.

IBM watsonx.ai

1. Complete the fields for **watsonx.ai API Endpoint**, **IBM Project ID**, and **IBM API key**. These values are found in your IBM watsonx deployment.
2. Under **Advanced settings**, select your **Language Model**.
3. Click **Complete**.
4. In the second onboarding panel, select a provider for embeddings and select your **Embedding Model**.
5. To complete the onboarding tasks, click **What is OpenRAG**, and then click **Add a Document**. Alternatively, click → **Skip overview**.

6. Continue with the Quickstart.

Ollama

> **INFO**
>
> Ollama isn't installed with OpenRAG. To install Ollama, see the Ollama documentation.

1. To connect to an Ollama server running on your local machine, enter your Ollama server's base URL address. The default Ollama server address is `http://localhost:11434`. OpenRAG connects to the Ollama server and populates the model lists with the server's available models.

2. Select the **Embedding Model** and **Language Model** your Ollama server is running.

   Ollama model selection and external server configuration

   Using Ollama for your OpenRAG language model provider offers greater flexibility and configuration, but can also be overwhelming to start. These recommendations are a reasonable starting point for users with at least one GPU and experience running LLMs locally.

   For best performance, OpenRAG recommends OpenAI's `gpt-oss:20b` language model. However, this model uses 16GB of RAM, so consider using Ollama Cloud or running Ollama on a remote machine.

   For generating embeddings, OpenRAG recommends the `nomic-embed-text` embedding model, which provides high-quality embeddings optimized for retrieval tasks.

   To run models in **Ollama Cloud**, follow these steps:

   i. Sign in to Ollama Cloud. In a terminal, enter `ollama signin` to connect your local environment with Ollama Cloud.
   ii. To run the model, in Ollama, select the `gpt-oss:20b-cloud` model, or run `ollama run gpt-oss:20b-cloud` in a terminal. Ollama Cloud models are run at the same URL as your local Ollama server at `http://localhost:11434`, and automatically offloaded to Ollama's cloud service.

iii. Connect OpenRAG to the same local Ollama server as you would for local models in onboarding, using the default address of `http://localhost:11434`.
iv. In the **Language model** field, select the `gpt-oss:20b-cloud` model.

To run models on a **remote Ollama server**, follow these steps:

i. Ensure your remote Ollama server is accessible from your OpenRAG instance.
ii. In the **Ollama Base URL** field, enter your remote Ollama server's base URL, such as `http://your-remote-server:11434`. OpenRAG connects to the remote Ollama server and populates the lists with the server's available models.
iii. Select your **Embedding model** and **Language model** from the available options.

3. Click **Complete**.

4. To complete the onboarding tasks, click **What is OpenRAG**, and then click **Add a Document**.

5. Continue with the Quickstart.

Exit the OpenRAG TUI

To exit the OpenRAG TUI, navigate to the main menu, and then press `q`. The OpenRAG containers continue to run until they are stopped. For more information, see Manage OpenRAG containers with the TUI .

To relaunch the TUI, run `uv run openrag`. If you installed OpenRAG with `uvx`, run `uvx openrag`.

Manage OpenRAG containers with the TUI

After installation, the TUI can deploy, manage, and upgrade your OpenRAG containers.

Diagnostics

The **Diagnostics** menu provides health monitoring for your container runtimes and monitoring of your OpenSearch security.

Status

The **Status** menu displays information on your container deployment. Here you can check container health, find your service ports, view logs, and upgrade your containers.

- **Logs**: To view streaming logs, select the container you want to view, and press `l`. To copy the logs, click **Copy to Clipboard**.
- **Upgrade**: Check for updates. For more information, see upgrade OpenRAG.
- **Reset**: This is a destructive action that resets your containers.
- **Native services**: From the **Status** menu, you can view the status, port, and process ID (PID) of the OpenRAG native services. You can also click **Stop** or **Restart** to stop and start OpenRAG native services.

  A *native service* in OpenRAG is a service that runs locally on your machine, not within a container. For example, the `docling serve` process is an OpenRAG native service because this document processing service runs on your local machine, separate from the OpenRAG containers.

Reset containers

> **WARNING**
> This is a destructive action that destroys and recreates all of your OpenRAG containers.

To destroy and recreate your OpenRAG containers, go to the TUI **Status** menu, and then click **Reset**.

The **Reset** function runs two commands. First, it stops and removes all containers, volumes, and local images:

```
docker compose down --volumes --remove-orphans --rmi local
```

Then, it removes any additional Docker objects with `docker system prune -f`.

If you reset your containers as part of reinstalling OpenRAG, continue the reinstallation process after resetting the containers.

Start all services

On the TUI main page, click **Start All Services** to start the OpenRAG containers and launch OpenRAG itself.

When you start all services, the following processes happen:

1. OpenRAG automatically detects your container runtime, and then checks if your machine has compatible GPU support by checking for `CUDA`, `NVIDIA_SMI`, and Docker/Podman runtime support. This check determines which Docker Compose file OpenRAG uses.

2. OpenRAG pulls the OpenRAG container images with `docker compose pull` if any images are missing.

3. OpenRAG deploys the containers with `docker compose up -d`.

Upgrade OpenRAG

To upgrade OpenRAG, upgrade the OpenRAG Python package, and then upgrade the OpenRAG containers.

This is a two part process because upgrading the OpenRAG Python package updates the TUI and Python code, but the container versions are controlled by environment variables in your `.env` file.

1. Stop your OpenRAG containers: In the OpenRAG TUI, go to the **Status** menu, and then click **Stop Services**.

2. Upgrade the OpenRAG Python package to the latest version from PyPI.

Automatic installer or uvx

Use these steps to upgrade the Python package if you installed OpenRAG using the automatic installer or `uvx`:

1. Navigate to your OpenRAG workspace directory:

```
cd openrag-workspace
```

2. Upgrade the OpenRAG package:

```
uvx --from openrag openrag
```

To upgrade to a specific version:

```
uvx --from openrag==0.1.33 openrag
```

Python project (uv add)

Use these steps to upgrade the Python package if you installed OpenRAG in a Python project with `uv add`:

1. Navigate to your project directory:

```
cd YOUR_PROJECT_NAME
```

2. Update OpenRAG to the latest version:

```
uv add --upgrade openrag
```

To upgrade to a specific version:

```
uv add --upgrade openrag==0.1.33
```

3. Start the OpenRAG TUI:

```
uv run openrag
```

Virtual environment (uv pip install)

Use these steps to upgrade the Python package if you installed OpenRAG in a venv with `uv pip install`:

1. Activate your virtual environment.

2. Upgrade OpenRAG:

   ```
   uv pip install --upgrade openrag
   ```

   To upgrade to a specific version:

   ```
   uv pip install --upgrade openrag==0.1.33
   ```

3. Start the OpenRAG TUI:

   ```
   uv run openrag
   ```

4. Start the upgraded OpenRAG containers: In the OpenRAG TUI, click **Start All Services**, and then wait while the containers start.

   After upgrading the Python package, OpenRAG runs `docker compose pull` to get the appropriate container images matching the version specified in your OpenRAG `.env` file. Then, it recreates the containers with the new images using `docker compose up -d --force-recreate`.

   In the `.env` file, the `OPENRAG_VERSION` environment variable is set to `latest` by default, which it pulls the `latest` available container images. To pin a specific container image version, you can set `OPENRAG_VERSION` to the desired container image version, such as `OPENRAG_VERSION=0.1.33`.

   However, when you upgrade the Python package, OpenRAG automatically attempts to keep the `OPENRAG_VERSION` synchronized with the Python package version. You might need to edit the `.env` file after upgrading the Python package to enforce a different container version. The TUI warns you if it detects a version mismatch.

   If you get an error that `langflow container already exists` error during upgrade, see Langflow container already exists during upgrade.

5. When the upgrade process is complete, you can close the **Status** window and continue using OpenRAG.

Reinstall OpenRAG

To reinstall OpenRAG with a completely fresh setup:

1. In the TUI **Status** menu, reset your containers to destroy the existing OpenRAG containers and their data.

2. Optional: Delete your project's `.env` file.

   The Reset operation doesn't remove your project's `.env` file, so your passwords, API keys, and OAuth settings can be preserved. If you delete the `.env` file, you must run the Set up OpenRAG with the TUI process again to create a new configuration file.

3. Optional: Delete your OpenSearch knowledge base by deleting the contents of the `./opensearch-data` folder in your OpenRAG installation directory.

4. In the TUI **Setup** menu, repeat the Basic Setup process:

   i. Click **Start All Services** to pull container images and start them.
   ii. Under **Native Services**, click **Start** to start the Docling service.
   iii. Click **Open App** to open the OpenRAG application.
   iv. Continue with application onboarding.

If reinstalling OpenRAG and deleting the `.env` file doesn't reset setup or onboarding, see Reinstalling OpenRAG doesn't reset onboarding.

Install OpenRAG containers

OpenRAG has two Docker Compose files. Both files deploy the same applications and containers locally, but they are for different environments:

- `docker-compose.yml` is an OpenRAG deployment with GPU support for accelerated AI processing. This Docker Compose file requires an NVIDIA GPU with CUDA support.

- `docker-compose-cpu.yml` is a CPU-only version of OpenRAG for systems without NVIDIA GPU support. Use this Docker Compose file for environments where GPU drivers aren't available.

Prerequisites

- Install the following:

    - Python version 3.13 or later.
    - uv.
    - Podman (recommended) or Docker.
    - `podman-compose` or Docker Compose. To use Docker Compose with Podman, you must alias Docker Compose commands to Podman commands.
- Microsoft Windows only: To run OpenRAG on Windows, you must use the Windows Subsystem for Linux (WSL).

Install WSL for OpenRAG

i. Install WSL with the Ubuntu distribution using WSL 2:

```
wsl --install -d Ubuntu
```

For new installations, the `wsl --install` command uses WSL 2 and Ubuntu by default.

For existing WSL installations, you can change the distribution and check the WSL version.

> **KNOWN LIMITATION**

> OpenRAG isn't compatible with nested virtualization, which can cause networking issues. Don't install OpenRAG on a WSL distribution that is installed inside a Windows VM. Instead, install OpenRAG on your base OS or a non-nested Linux VM.

ii. Start your WSL Ubuntu distribution if it doesn't start automatically.

iii. Set up a username and password for your WSL distribution.

iv. Install Docker Desktop for Windows with WSL 2. When you reach the Docker Desktop **WSL integration** settings, make sure your Ubuntu distribution is enabled, and then click **Apply & Restart** to enable Docker support in WSL.

v. Install and run OpenRAG from within your WSL Ubuntu distribution.

If you encounter issues with port forwarding or the Windows Firewall, you might need to adjust the Hyper-V firewall settings to allow communication between your WSL distribution and the Windows host. For more troubleshooting advice for networking issues, see Troubleshooting WSL common issues.

- Prepare model providers and credentials.

  During application onboarding, you must select language model and embedding model providers. If your chosen provider offers both types, you can use the same provider for both selections. If your provider offers only one type, such as Anthropic, you must select two providers.

  Gather the credentials and connection details for your chosen model providers before starting onboarding:

  - OpenAI: Create an OpenAI API key.
  - Anthropic language models: Create an Anthropic API key.
  - IBM watsonx.ai: Get your watsonx.ai API endpoint, IBM project ID, and IBM API key from your watsonx deployment.
  - Ollama: Use the Ollama documentation to set up your Ollama instance locally, in the cloud, or on a remote server, and then get your Ollama server's base URL.
- Optional: Install GPU support with an NVIDIA GPU, CUDA support, and compatible NVIDIA drivers on the OpenRAG host machine. This is required to use the GPU-

accelerated Docker Compose file. If you choose not to use GPU support, you must use the CPU-only Docker Compose file instead.

Install OpenRAG with Docker Compose

To install OpenRAG with Docker Compose, do the following:

1. Clone the OpenRAG repository.

   ```
   git clone https://github.com/langflow-ai/openrag.git
   cd openrag
   ```

2. Install dependencies.

   ```
   uv sync
   ```

3. Copy the example `.env` file included in the repository root. The example file includes all environment variables with comments to guide you in finding and setting their values.

   ```
   cp .env.example .env
   ```

   Alternatively, create a new `.env` file in the repository root.

   ```
   touch .env
   ```

4. The Docker Compose files are populated with the values from your `.env` file. The `OPENSEARCH_PASSWORD` value must be set. `OPENSEARCH_PASSWORD` can be automatically generated when using the TUI, but for a Docker Compose installation, you can set it manually instead. To generate an OpenSearch admin password, see the OpenSearch documentation.

   The following values are optional:

   ```
   ```

```
OPENAI_API_KEY=your_openai_api_key
LANGFLOW_SECRET_KEY=your_secret_key
```

`OPENAI_API_KEY` is optional. You can provide it during application onboarding or choose a different model provider. If you want to set it in your `.env` file, you can find your OpenAI API key in your OpenAI account.

`LANGFLOW_SECRET_KEY` is optional. Langflow will auto-generate it if not set. For more information, see the Langflow documentation.

The following Langflow configuration values are optional but important to consider:

```
LANGFLOW_SUPERUSER=admin
LANGFLOW_SUPERUSER_PASSWORD=your_langflow_password
```

`LANGFLOW_SUPERUSER` defaults to `admin`. You can omit it or set it to a different username. `LANGFLOW_SUPERUSER_PASSWORD` is optional. If omitted, Langflow runs in autologin mode with no password required. If set, Langflow requires password authentication.

For more information on configuring OpenRAG with environment variables, see Environment variables.

5. Start `docling serve` on the host machine. OpenRAG Docker installations require that `docling serve` is running on port 5001 on the host machine. This enables Mac MLX support for document processing.

```
uv run python scripts/docling_ctl.py start --port 5001
```

6. Confirm `docling serve` is running.

```
uv run python scripts/docling_ctl.py status
```

Make sure the response shows that `docling serve` is running, for example:

```
```

```
Status: running
Endpoint: http://127.0.0.1:5001
Docs: http://127.0.0.1:5001/docs
PID: 27746
```

7. Deploy OpenRAG locally with Docker Compose based on your deployment type.

docker-compose.yml

```
docker compose build
docker compose up -d
```

docker-compose-cpu.yml

```
docker compose -f docker-compose-cpu.yml up -d
```

The OpenRAG Docker Compose file starts five containers:

Container Name	Default Address	Purpose
OpenRAG Backend	http://localhost:8000	FastAPI server and core functionality.
OpenRAG Frontend	http://localhost:3000	React web interface for users.
Langflow	http://localhost:7860	AI workflow engine and flow management.
OpenSearch	http://localhost:9200	Vector database for document storage.
OpenSearch Dashboards	http://localhost:5601	Database administration interface.

8. Verify installation by confirming all services are running.

```
docker compose ps
```

You can now access OpenRAG at the following endpoints:

- **Frontend**: http://localhost:3000
- **Backend API**: http://localhost:8000
- **Langflow**: http://localhost:7860

9. Continue with application onboarding.

To stop `docling serve` when you're done with your OpenRAG deployment, run:

```
uv run python scripts/docling_ctl.py stop
```

Application onboarding

The first time you start OpenRAG, regardless of how you installed it, you must complete application onboarding.

Some of these variables, such as the embedding models, can be changed seamlessly after onboarding. Others are immutable and require you to destroy and recreate the OpenRAG containers. For more information, see Environment variables.

You can use different providers for your language model and embedding model, such as Anthropic for the language model and OpenAI for the embeddings model. Additionally, you can set multiple embedding models.

You only need to complete onboarding for your preferred providers.

Anthropic

> **INFO**
>
> Anthropic doesn't provide embedding models. If you select Anthropic for your language model, you must select a different provider for embeddings.

1. Enable **Use environment Anthropic API key** to automatically use your key from the `.env` file. Alternatively, paste an Anthropic API key into the field.
2. Under **Advanced settings**, select your **Language Model**.
3. Click **Complete**.
4. In the second onboarding panel, select a provider for embeddings and select your **Embedding Model**.

5. To complete the onboarding tasks, click **What is OpenRAG**, and then click **Add a Document**. Alternatively, click → **Skip overview**.
6. Continue with the Quickstart.

OpenAI

1. Enable **Get API key from environment variable** to automatically enter your key from the TUI-generated `.env` file. Alternatively, paste an OpenAI API key into the field.
2. Under **Advanced settings**, select your **Language Model**.
3. Click **Complete**.
4. In the second onboarding panel, select a provider for embeddings and select your **Embedding Model**.
5. To complete the onboarding tasks, click **What is OpenRAG**, and then click **Add a Document**. Alternatively, click → **Skip overview**.
6. Continue with the Quickstart.

IBM watsonx.ai

1. Complete the fields for **watsonx.ai API Endpoint**, **IBM Project ID**, and **IBM API key**. These values are found in your IBM watsonx deployment.
2. Under **Advanced settings**, select your **Language Model**.
3. Click **Complete**.
4. In the second onboarding panel, select a provider for embeddings and select your **Embedding Model**.
5. To complete the onboarding tasks, click **What is OpenRAG**, and then click **Add a Document**. Alternatively, click → **Skip overview**.
6. Continue with the Quickstart.

Ollama

> **INFO**
>
> Ollama isn't installed with OpenRAG. To install Ollama, see the Ollama documentation.

1. To connect to an Ollama server running on your local machine, enter your Ollama server's base URL address. The default Ollama server address is

`http://localhost:11434`. OpenRAG connects to the Ollama server and populates the model lists with the server's available models.

2. Select the **Embedding Model** and **Language Model** your Ollama server is running.

Ollama model selection and external server configuration

Using Ollama for your OpenRAG language model provider offers greater flexibility and configuration, but can also be overwhelming to start. These recommendations are a reasonable starting point for users with at least one GPU and experience running LLMs locally.

For best performance, OpenRAG recommends OpenAI's `gpt-oss:20b` language model. However, this model uses 16GB of RAM, so consider using Ollama Cloud or running Ollama on a remote machine.

For generating embeddings, OpenRAG recommends the `nomic-embed-text` embedding model, which provides high-quality embeddings optimized for retrieval tasks.

To run models in **Ollama Cloud**, follow these steps:

i. Sign in to Ollama Cloud. In a terminal, enter `ollama signin` to connect your local environment with Ollama Cloud.
ii. To run the model, in Ollama, select the `gpt-oss:20b-cloud` model, or run `ollama run gpt-oss:20b-cloud` in a terminal. Ollama Cloud models are run at the same URL as your local Ollama server at `http://localhost:11434`, and automatically offloaded to Ollama's cloud service.
iii. Connect OpenRAG to the same local Ollama server as you would for local models in onboarding, using the default address of `http://localhost:11434`.
iv. In the **Language model** field, select the `gpt-oss:20b-cloud` model.

To run models on a **remote Ollama server**, follow these steps:

i. Ensure your remote Ollama server is accessible from your OpenRAG instance.
ii. In the **Ollama Base URL** field, enter your remote Ollama server's base URL, such as `http://your-remote-server:11434`. OpenRAG connects to the remote Ollama server and populates the lists with the server's available models.

iii. Select your **Embedding model** and **Language model** from the available options.

3. Click **Complete**.

4. To complete the onboarding tasks, click **What is OpenRAG**, and then click **Add a Document**.

5. Continue with the Quickstart.

Container management commands

Manage your OpenRAG containers with the following commands. These commands are also available in the TUI's Status menu.

Upgrade containers

Upgrade your containers to the latest version while preserving your data.

```
docker compose pull
docker compose up -d --force-recreate
```

Rebuild containers (destructive)

Reset state by rebuilding all of your containers. Your OpenSearch and Langflow databases will be lost. Documents stored in the `./openrag-documents` directory will persist, since the directory is mounted as a volume in the OpenRAG backend container.

```
docker compose up --build --force-recreate --remove-orphans
```

Remove all containers and data (destructive)

Completely remove your OpenRAG installation and delete all data. This deletes all of your data, including OpenSearch data, uploaded documents, and authentication.

```
docker compose down --volumes --remove-orphans --rmi local
docker system prune -f
```

Use Langflow in OpenRAG

OpenRAG includes a built-in Langflow instance for creating and managing functional application workflows called *flows*. In a flow, the individual workflow steps are represented by *components* that are connected together to form a complete process.

OpenRAG includes several built-in flows:

- The **OpenRAG OpenSearch Agent** flow powers the **Chat** feature in OpenRAG.
- The **OpenSearch Ingestion** and **OpenSearch URL Ingestion** flows process documents and web content for storage in your OpenSearch knowledge base.
- The **OpenRAG OpenSearch Nudges** flow provides optional contextual suggestions in the OpenRAG **Chat**.

You can customize these flows and create your own flows using OpenRAG's embedded Langflow visual editor.

Inspect and modify flows

All OpenRAG flows are designed to be modular, performant, and provider-agnostic.

To modify a flow in OpenRAG, click **Settings**. From here, you can quickly edit commonly used parameters, such as the **Language model** and **Agent Instructions**. To further explore and edit the flow, click **Edit in Langflow** to launch the embedded Langflow visual editor where you can fully customize the flow to suit your use case.

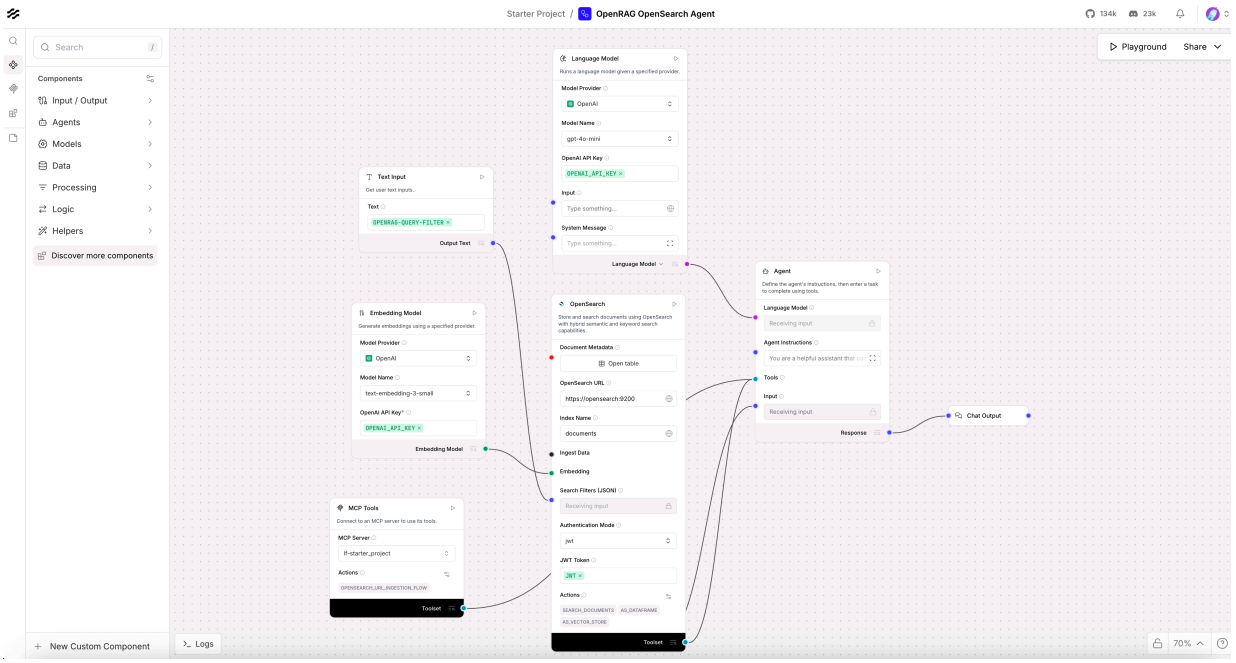
For example, to view and edit the built-in **Chat** flow (the **OpenRAG OpenSearch Agent** flow), do the following:

1. In OpenRAG, click **Chat**.
2. Click **Settings**, and then click **Edit in Langflow** to launch the Langflow visual editor in a new browser window.

   If prompted to acknowledge that you are entering Langflow, click **Proceed**.

   If Langflow requests login information, enter the `LANGFLOW_SUPERUSER` and `LANGFLOW_SUPERUSER_PASSWORD` from the `.env` file in your OpenRAG installation

directory.



3. Modify the flow as desired, and then press `Command` + `S` (`Ctrl` + `S`) to save your changes.

   You can close the Langflow browser window, or leave it open if you want to continue experimenting with the flow editor.

   > **TIP**
   >
   > If you modify the built-in **Chat** flow, make sure you click + in the **Conversations** tab to start a new conversation. This ensures that the chat doesn't persist any context from the previous conversation with the original flow settings.

Revert a built-in flow to its original configuration

After you edit a built-in flow, you can click **Restore flow** on the **Settings** page to revert the flow to its original state when you first installed OpenRAG. This is a destructive action that discards all customizations to the flow.

Build custom flows and use other Langflow functionality

In addition to OpenRAG's built-in flows, all Langflow features are available through OpenRAG, including the ability to create your own flows and popular extensibility features

such as the following:

- Create custom components.
- Integrate with many third-party services through bundles.
- Use MCP clients and MCP servers, and serve flows as MCP tools for your agentic flows.

Explore the Langflow documentation to learn more about the Langflow platform, features, and visual editor.

Set the Langflow version

By default, OpenRAG is pinned to the latest Langflow Docker image for stability.

If necessary, you can set a specific Langflow version with the `LANGFLOW_VERSION`. However, there are risks to changing this setting:

- The Langflow documentation describes the functionality present in the latest release of the Langflow OSS Python package. If your `LANGFLOW_VERSION` is different, the Langflow documentation might not align with the features and default settings in your OpenRAG installation.
- Components might break, including components in OpenRAG's built-in flows.
- Default settings and behaviors might change causing unexpected results when OpenRAG expects a newer default.

Configure knowledge

OpenRAG includes a built-in OpenSearch instance that serves as the underlying datastore for your *knowledge* (documents). This specialized database is used to store and retrieve your documents and the associated vector data (embeddings).

The documents in your OpenSearch knowledge base provide specialized context in addition to the general knowledge available to the language model that you select when you install OpenRAG or edit a flow.

You can upload documents from a variety of sources to populate your knowledge base with unique content, such as your own company documents, research papers, or websites. Documents are processed through OpenRAG's knowledge ingestion flows with Docling.

Then, the OpenRAG **Chat** can run similarity searches against your OpenSearch database to retrieve relevant information and generate context-aware responses.

You can configure how documents are ingested and how the **Chat** interacts with your knowledge base.

Browse knowledge

The **Knowledge** page lists the documents OpenRAG has ingested into your OpenSearch database, specifically in an OpenSearch index named `documents`.

To explore the raw contents of your knowledge base, click **Knowledge** to get a list of all ingested documents. Click a document to view the chunks produced from splitting the document during ingestion.

OpenRAG includes some initial documents about OpenRAG. You can use these documents to ask OpenRAG about itself, and to test the **Chat** feature before uploading your own documents. If you delete these documents, you won't be able to ask OpenRAG about itself and it's own functionality. It is recommended that you keep these documents, and use filters to separate them from your other knowledge.

OpenSearch authentication and document access

When you install OpenRAG, you can choose between two setup modes: **Basic Setup** and **Advanced Setup**. The mode you choose determines how OpenRAG authenticates with OpenSearch and controls access to documents:

- **Basic Setup (no-auth mode)**: If you choose **Basic Setup**, then OpenRAG is installed in no-auth mode. This mode uses one, anonymous JWT token for OpenSearch authentication. There is no differentiation between users. All users that access your OpenRAG instance can access all documents uploaded to your OpenSearch knowledge base.
- **Advanced Setup (OAuth mode)**: If you choose **Advanced Setup**, then OpenRAG is installed in OAuth mode. This mode uses a unique JWT token for each OpenRAG user, and each document is tagged with user ownership. Documents are filtered by user owner. This means users see only the documents that they uploaded or have access to.

You can enable OAuth mode after installation. For more information, see Ingest files with OAuth connectors.

OpenSearch indexes

An OpenSearch index is a collection of documents in an OpenSearch database.

By default, all documents you upload to your OpenRAG knowledge base are stored in an index named `documents`.

It is possible to change the index name by editing the ingestion flow. However, this can impact dependent processes, such as the filters and **Chat** flow, that reference the `documents` index by default. Make sure you edit other flows as needed to ensure all processes use the same index name.

If you encounter errors or unexpected behavior after changing the index name, you can revert the flows to their original configuration, or delete knowledge to clear the existing documents from your knowledge base.

Knowledge ingestion settings

> **WARNING**

> Knowledge ingestion settings apply to documents you upload after making the changes. Documents uploaded before changing these settings aren't reprocessed.

After changing knowledge ingestion settings, you must determine if you need to reupload any documents to be consistent with the new settings.

It isn't always necessary to reupload documents after changing knowledge ingestion settings. For example, it is typical to upload some documents with OCR enabled and others without OCR enabled.

If needed, you can use filters to separate documents that you uploaded with different settings, such as different embedding models.

Set the embedding model and dimensions

When you install OpenRAG, you select at least one embedding model during application onboarding. OpenRAG automatically detects and configures the appropriate vector dimensions for your selected embedding model, ensuring optimal search performance and compatibility.

In the OpenRAG repository, you can find the complete list of supported models in `models_service.py` and the corresponding vector dimensions in `settings.py`.

During application onboarding, you can select from the supported models. The default embedding dimension is `1536`, and the default model is the OpenAI `text-embedding-3-small`.

If you want to use an unsupported model, you must manually set the model in your OpenRAG configuration. If you use an unsupported embedding model that doesn't have defined dimensions in `settings.py`, then OpenRAG falls back to the default dimensions (1536) and logs a warning. OpenRAG's OpenSearch instance and flows continue to work, but similarity search quality can be affected if the actual model dimensions aren't 1536.

To change the embedding model after onboarding, it is recommended that you modify the embedding model setting in the OpenRAG **Settings** page or in your OpenRAG configuration. This will automatically update all relevant OpenRAG flows to use the new embedding model configuration.

Set Docling parameters

OpenRAG uses Docling for document ingestion because it supports many file formats, processes tables and images well, and performs efficiently.

When you upload documents, Docling processes the files, splits them into chunks, and stores them as separate, structured documents in your OpenSearch knowledge base.

You can use either Docling Serve or OpenRAG's built-in Docling ingestion pipeline to process documents.

Docling Serve ingestion

By default, OpenRAG uses Docling Serve. This means that OpenRAG starts a `docling serve` process on your local machine and runs Docling ingestion through an API service.

Built-in Docling ingestion

If you want to use OpenRAG's built-in Docling ingestion pipeline instead of the separate Docling Serve service, set `DISABLE_INGEST_WITH_LANGFLOW=true` in your OpenRAG environment variables.

The built-in pipeline uses the Docling processor directly instead of through the Docling Serve API.

For the underlying functionality, see `processors.py` in the OpenRAG repository.

To modify the Docling ingestion and embedding parameters, click **Settings** in the OpenRAG user interface.

> **TIP**
> OpenRAG warns you if `docling serve` isn't running. You can start and stop OpenRAG services from the TUI main menu with **Start Native Services** or **Stop Native Services**.

- **Embedding model**: Select the model to use to generate vector embeddings for your documents.

This is initially set during installation. The recommended way to change this setting is in the OpenRAG **Settings** or your OpenRAG configuration. This will automatically update all relevant OpenRAG flows to use the new embedding model configuration.

If you uploaded documents prior to changing the embedding model, you can create filters to separate documents embedded with different models, or you can reupload all documents to regenerate embeddings with the new model. If you want to use multiple embeddings models, similarity search (in the **Chat**) can take longer as it searching each model's embeddings separately.

- **Chunk size**: Set the number of characters for each text chunk when breaking down a file. Larger chunks yield more context per chunk, but can include irrelevant information. Smaller chunks yield more precise semantic search, but can lack context. The default value is 1000 characters, which is usually a good balance between context and precision.

- **Chunk overlap**: Set the number of characters to overlap over chunk boundaries. Use larger overlap values for documents where context is most important. Use smaller overlap values for simpler documents or when optimization is most important. The default value is 200 characters, which represents an overlap of 20 percent if the **Chunk size** is 1000. This is suitable for general use. For faster processing, decrease the overlap to approximately 10 percent. For more complex documents where you need to preserve context across chunks, increase it to approximately 40 percent.

- **Table Structure**: Enables Docling's `DocumentConverter` tool for parsing tables. Instead of treating tables as plain text, tables are output as structured table data with preserved relationships and metadata. This option is enabled by default.

- **OCR**: Enables Optical Character Recognition (OCR) processing when extracting text from images and ingesting scanned documents. This setting is best suited for processing text-based documents faster with Docling's `DocumentConverter`. Images are ignored and not processed.

  This option is disabled by default. Enabling OCR can slow ingestion performance.

  If OpenRAG detects that the local machine is running on macOS, OpenRAG uses the ocrmac OCR engine. Other platforms use easyocr.

- **Picture descriptions**: Only applicable if **OCR** is enabled. Adds image descriptions generated by the `SmolVLM-256M-Instruct` model. Enabling picture descriptions can slow ingestion performance.

Set the local documents path

The default path for local uploads is the `./openrag-documents` subdirectory in your OpenRAG installation directory. This is mounted to the `/app/openrag-documents/` directory inside the OpenRAG container. Files added to the host or container directory are visible in both locations.

To change this location, modify the **Documents Paths** variable in either the **Advanced Setup** menu or in the `.env` used by Docker Compose.

Delete knowledge

To clear your entire knowledge base, delete the contents of the `./opensearch-data` folder in your OpenRAG installation directory. This is a destructive operation that cannot be undone.

See also

- Ingest knowledge
- Filter knowledge
- Chat with knowledge
- Inspect and modify flows

Ingest knowledge

Upload documents to your OpenRAG OpenSearch instance to populate your knowledge base with unique content, such as your own company documents, research papers, or websites. Documents are processed through OpenRAG's knowledge ingestion flows with Docling.

OpenRAG can ingest knowledge from direct file uploads, URLs, and OAuth authenticated connectors.

Knowledge ingestion is powered by OpenRAG's built-in knowledge ingestion flows that use Docling to process documents before storing the documents in your OpenSearch database. During ingestion, documents are broken into smaller chunks of content that are then embedded using your selected embedding model. Then, the chunks, embeddings, and associated metadata (which connects chunks of the same document) are stored in your OpenSearch database.

To modify chunking behavior and other ingestion settings, see Knowledge ingestion settings and Inspect and modify flows.

Ingest local files and folders

You can upload files and folders from your local machine to your knowledge base:

1. Click **Knowledge** to view your OpenSearch knowledge base.
2. Click **Add Knowledge** to add your own documents to your OpenRAG knowledge base.
3. To upload one file, click **File**. To upload all documents in a folder, click **Folder**.

   The default path is the `./documents` subdirectory in your OpenRAG installation directory. To change this path, see Set the local documents path.

The selected files are processed in the background through the **OpenSearch Ingestion** flow.

About the OpenSearch Ingestion flow

When you upload documents locally or with OAuth connectors, the **OpenSearch Ingestion** flow runs in the background. By default, this flow uses Docling Serve to import and process documents.

Like all OpenRAG flows, you can inspect the flow in Langflow, and you can customize it if you want to change the knowledge ingestion settings.

The **OpenSearch Ingestion** flow is comprised of several components that work together to process and store documents in your knowledge base:

- **Docling Serve** component: Ingests files and processes them by connecting to OpenRAG's local Docling Serve service. The output is `DoclingDocument` data that contains the extracted text and metadata from the documents.
- **Export DoclingDocument** component: Exports processed `DoclingDocument` data to Markdown format with image placeholders. This conversion standardizes the document data in preparation for further processing.
- **DataFrame Operations** component: Three of these components run sequentially to add metadata to the document data: `filename`, `file_size`, and `mimetype`.
- **Split Text** component: Splits the processed text into chunks, based on the configured chunk size and overlap settings.
- **Secret Input** component: If needed, four of these components securely fetch the OAuth authentication configuration variables: `CONNECTOR_TYPE`, `OWNER`, `OWNER_EMAIL`, and `OWNER_NAME`.
- **Create Data** component: Combines the authentication credentials from the **Secret Input** components into a structured data object that is associated with the document embeddings.
- **Embedding Model** component: Generates vector embeddings using your selected embedding model.
- **OpenSearch** component: Stores the processed documents and their embeddings in a `documents` index of your OpenRAG OpenSearch knowledge base.

The default address for the OpenSearch instance is `https://opensearch:9200`. To change this address, edit the `OPENSEARCH_PORT` environment variable.

The default authentication method is JSON Web Token (JWT) authentication. If you edit the flow, you can select `basic` auth mode, which uses the `OPENSEARCH_USERNAME` and `OPENSEARCH_PASSWORD` environment variables for authentication instead of JWT.

You can monitor ingestion to see the progress of the uploads and check for failed uploads.

Ingest local files temporarily

When using the OpenRAG **Chat**, click + in the chat input field to upload a file to the current chat session. Files added this way are processed and made available to the agent for the current conversation only. These files aren't stored in the knowledge base permanently.

Ingest files with OAuth connectors

OpenRAG can use OAuth authenticated connectors to ingest documents from the following external services:

- AWS S3
- Google Drive
- Microsoft OneDrive
- Microsoft Sharepoint

These connectors enable seamless ingestion of files from cloud storage to your OpenRAG knowledge base.

Individual users can connect their personal cloud storage accounts to OpenRAG. Each user must separately authorize OpenRAG to access their own cloud storage. When a user connects a cloud storage service, they are redirected to authenticate with that service provider and grant OpenRAG permission to sync documents from their personal cloud storage.

Enable OAuth connectors

Before users can connect their own cloud storage accounts, you must configure the provider's OAuth credentials in OpenRAG. Typically, this requires that you register OpenRAG as an OAuth application in your cloud provider, and then obtain the app's OAuth credentials, such as a client ID and secret key. To enable multiple connectors, you must register an app and generate credentials for each provider.

TUI Advanced Setup

If you use the TUI to manage your OpenRAG containers, provide OAuth credentials in the **Advanced Setup**.

You can do this during installation, or you can add the credentials afterwards:

1. If OpenRAG is running, stop it: Go to **Status**, and then click **Stop Services**.

2. Click **Advanced Setup**, and then add the OAuth credentials for the cloud storage providers that you want to use:

    - **Amazon**: Provide your AWS Access Key ID and AWS Secret Access Key with access to your S3 instance. For more information, see the AWS documentation on Configuring access to AWS applications.
    - **Google**: Provide your Google OAuth Client ID and Google OAuth Client Secret. You can generate these in the Google Cloud Console. For more information, see the Google OAuth client documentation.
    - **Microsoft**: For the Microsoft OAuth Client ID and Microsoft OAuth Client Secret, provide Azure application registration credentials for SharePoint and OneDrive. For more information, see the Microsoft Graph OAuth client documentation.
3. The OpenRAG TUI presents redirect URIs for your OAuth app that you must register with your OAuth provider. These are the URLs your OAuth provider will redirect back to after users authenticate and grant access to their cloud storage.

4. Click **Save Configuration**.

    OpenRAG regenerates the `.env` file with the given credentials.

5. Click **Start Container Services**.

Docker Compose .env file

If you install OpenRAG with self-managed containers, set OAuth credentials in the `.env` file for Docker Compose.

You can do this during initial set up, or you can add the credentials afterwards:

1. Stop your OpenRAG deployment.

   Podman

   ```
   podman stop --all
   ```

   Docker

   ```
   docker stop $(docker ps -q)
   ```

2. Edit the `.env` file for Docker Compose to add the OAuth credentials for the cloud storage providers that you want to use:

   - **Amazon**: Provide your AWS Access Key ID and AWS Secret Access Key with access to your S3 instance. For more information, see the AWS documentation on Configuring access to AWS applications.

     ```
     AWS_ACCESS_KEY_ID=
     AWS_SECRET_ACCESS_KEY=
     ```

   - **Google**: Provide your Google OAuth Client ID and Google OAuth Client Secret. You can generate these in the Google Cloud Console. For more information, see the Google OAuth client documentation.

     ```
     GOOGLE_OAUTH_CLIENT_ID=
     GOOGLE_OAUTH_CLIENT_SECRET=
     ```

   - **Microsoft**: For the Microsoft OAuth Client ID and Microsoft OAuth Client Secret, provide Azure application registration credentials for SharePoint and OneDrive. For more information, see the Microsoft Graph OAuth client documentation.

```
MICROSOFT_GRAPH_OAUTH_CLIENT_ID=
MICROSOFT_GRAPH_OAUTH_CLIENT_SECRET=
```

3. Save the `.env` file.

4. Restart your OpenRAG deployment:

   Podman

   ```
   podman-compose up -d
   ```

   Docker

   ```
   docker-compose up -d
   ```

Authenticate and ingest files from cloud storage

After you start OpenRAG with OAuth connectors enabled, each user is prompted to authenticate with the OAuth provider upon accessing your OpenRAG instance. Individual authentication is required to access a user's cloud storage from your OpenRAG instance. For example, if a user navigates to the default OpenRAG URL at `http://localhost:3000`, they are redirected to the OAuth provider's sign-in page. After authenticating and granting the required permissions for OpenRAG, the user is redirected back to OpenRAG.

To ingest knowledge with an OAuth connector, do the following:

1. Click **Knowledge** to view your OpenSearch knowledge base.

2. Click **Add Knowledge**, and then select a storage provider.

3. On the **Add Cloud Knowledge** page, click **Add Files**, and then select the files and folders to ingest from the connected storage.

4. Click **Ingest Files**.

The selected files are processed in the background through the **OpenSearch Ingestion** flow.

About the OpenSearch Ingestion flow

When you upload documents locally or with OAuth connectors, the **OpenSearch Ingestion** flow runs in the background. By default, this flow uses Docling Serve to import and process documents.

Like all OpenRAG flows, you can inspect the flow in Langflow, and you can customize it if you want to change the knowledge ingestion settings.

The **OpenSearch Ingestion** flow is comprised of several components that work together to process and store documents in your knowledge base:

- **Docling Serve** component: Ingests files and processes them by connecting to OpenRAG's local Docling Serve service. The output is `DoclingDocument` data that contains the extracted text and metadata from the documents.
- **Export DoclingDocument** component: Exports processed `DoclingDocument` data to Markdown format with image placeholders. This conversion standardizes the document data in preparation for further processing.
- **DataFrame Operations** component: Three of these components run sequentially to add metadata to the document data: `filename`, `file_size`, and `mimetype`.
- **Split Text** component: Splits the processed text into chunks, based on the configured chunk size and overlap settings.
- **Secret Input** component: If needed, four of these components securely fetch the OAuth authentication configuration variables: `CONNECTOR_TYPE`, `OWNER`, `OWNER_EMAIL`, and `OWNER_NAME`.
- **Create Data** component: Combines the authentication credentials from the **Secret Input** components into a structured data object that is associated with the document embeddings.
- **Embedding Model** component: Generates vector embeddings using your selected embedding model.
- **OpenSearch** component: Stores the processed documents and their embeddings in a `documents` index of your OpenRAG OpenSearch knowledge base.

The default address for the OpenSearch instance is `https://opensearch:9200`. To change this address, edit the `OPENSEARCH_PORT` environment variable.

The default authentication method is JSON Web Token (JWT) authentication. If you edit the flow, you can select `basic` auth mode, which uses the `OPENSEARCH_USERNAME` and `OPENSEARCH_PASSWORD` environment variables for authentication instead of JWT.

You can monitor ingestion to see the progress of the uploads and check for failed uploads.

Ingest knowledge from URLs

The **OpenSearch URL Ingestion** flow is used to ingest web content from URLs. This flow isn't directly accessible from the OpenRAG user interface. Instead, this flow is called by the **OpenRAG OpenSearch Agent** flow as a Model Context Protocol (MCP) tool. The agent can call this component to fetch web content from a given URL, and then ingest that content into your OpenSearch knowledge base.

Like all OpenRAG flows, you can inspect the flow in Langflow, and you can customize it.

For more information about MCP in Langflow, see the Langflow documentation on MCP clients and MCP servers.

Monitor ingestion

Document ingestion tasks run in the background.

In the OpenRAG user interface, a badge is shown on **Tasks** when OpenRAG tasks are active. Click **Tasks** to inspect and cancel tasks:

- **Active Tasks**: All tasks that are **Pending**, **Running**, or **Processing**. For each active task, depending on its state, you can find the task ID, start time, duration, number of files processed, and the total files enqueued for processing.
- **Pending**: The task is queued and waiting to start.
- **Running**: The task is actively processing files.

- **Processing**: The task is performing ingestion operations.
- **Failed**: Something went wrong during ingestion, or the task was manually canceled. For troubleshooting advice, see Troubleshoot ingestion.

To stop an active task, click ✕ **Cancel**. Canceling a task stops processing immediately and marks the task as **Failed**.

Ingestion performance expectations

The following performance test was conducted with Docling Serve.

On a local VM with 7 vCPUs and 8 GiB RAM, OpenRAG ingested approximately 5.03 GB across 1,083 files in about 42 minutes. This equates to approximately 2.4 documents per second.

You can generally expect equal or better performance on developer laptops, and significantly faster performance on servers. Throughput scales with CPU cores, memory, storage speed, and configuration choices, such as the embedding model, chunk size, overlap, and concurrency.

This test returned 12 error, approximately 1.1 percent of the total files ingested. All errors were file-specific, and they didn't stop the pipeline.

Ingestion performance test details

- Ingestion dataset:
    - Total files: 1,083 items mounted
    - Total size on disk: 5,026,474,862 bytes (approximately 5.03 GB)
- Hardware specifications:
    - Machine: Apple M4 Pro
    - Podman VM:
        - Name: podman-machine-default
        - Type: applehv
        - vCPUs: 7
        - Memory: 8 GiB
        - Disk size: 100 GiB

- Test results:

```
2025-09-24T22:40:45.542190Z /app/src/main.py:231 Ingesting
default documents when ready disable_langflow_ingest=False
2025-09-24T22:40:45.546385Z /app/src/main.py:270 Using Langflow
ingestion pipeline for default documents file_count=1082
...
2025-09-24T23:19:44.866365Z /app/src/main.py:351 Langflow
ingestion completed success_count=1070 error_count=12
total_files=1082
```

- Elapsed time: Approximately 42 minutes 15 seconds (2,535 seconds)
- Throughput: Approximately 2.4 documents per second

Troubleshoot ingestion

If an ingestion task fails, do the following:

- Make sure you are uploading supported file types.
- Split excessively large files into smaller files before uploading.
- Remove unusual embedded content, such as videos or animations, before uploading. Although Docling can replace some non-text content with placeholders during ingestion, some embedded content might cause errors.

If the OpenRAG **Chat** doesn't seem to use your documents correctly, browse your knowledge base to confirm that the documents are uploaded in full, and the chunks are correct.

If the documents are present and well-formed, check your knowledge filters. If a global filter is applied, make sure the expected documents are included in the global filter. If the global filter excludes any documents, the agent cannot access those documents unless you apply a chat-level filter or change the global filter.

If text is missing or incorrectly processed, you need to reupload the documents after modifying the ingestion parameters or the documents themselves. For example:

- Break combined documents into separate files for better metadata context.

- Make sure scanned documents are legible enough for extraction, and enable the **OCR** option. Poorly scanned documents might require additional preparation or rescanning before ingestion.
- Adjust the **Chunk Size** and **Chunk Overlap** settings to better suit your documents. Larger chunks provide more context but can include irrelevant information, while smaller chunks yield more precise semantic search but can lack context.

For more information about modifying ingestion parameters and flows, see Knowledge ingestion settings.

See also

- Configure knowledge
- Filter knowledge
- Chat with knowledge
- Inspect and modify flows

Filter knowledge

OpenRAG's knowledge filters help you organize and manage your knowledge base by creating pre-defined views of your documents.

Each knowledge filter captures a specific subset of documents based on given a search query and filters.

Knowledge filters can be used with different OpenRAG functionality. For example, knowledge filters can help agents access large knowledge bases efficiently by narrowing the scope of documents that you want the agent to use.

Create a filter

To create a knowledge filter, do the following:

1. Click **Knowledge**, and then click + **Knowledge Filters**.
2. Enter a **Name** and **Description**, and then click **Create Filter**.

   By default, new filters match all documents in your knowledge base. Modify the filter to customize it.

3. To modify the filter, click **Knowledge**, and then click your new filter. You can edit the following settings:

   - **Search Query**: Enter text for semantic search, such as `financial reports from Q4`.
   - **Data Sources**: Select specific data sources or folders to include.
   - **Document Types**: Filter by file type.
   - **Owners**: Filter by the user that uploaded the documents.
   - **Connectors**: Filter by upload source, such as the local file system or a Google Drive OAuth connector.
   - **Response Limit**: Set the maximum number of results to return from the knowledge base. The default is `10`.
   - **Score Threshold**: Set the minimum relevance score for similarity search. The default score is `0`.
4. To save your changes, click **Update Filter**.

Apply a filter

- **Apply a global filter**: Click **Knowledge**, and then enable the toggle next to your preferred filter. Only one filter can be the global filter. The global filter applies to all chat sessions.
- **Apply a chat filter**: In the **Chat** window, click **Filter**, and then select the filter to apply. Chat filters apply to one chat session only.

Delete a filter

1. Click **Knowledge**.
2. Click the filter that you want to delete.
3. Click **Delete Filter**.

Chat in OpenRAG

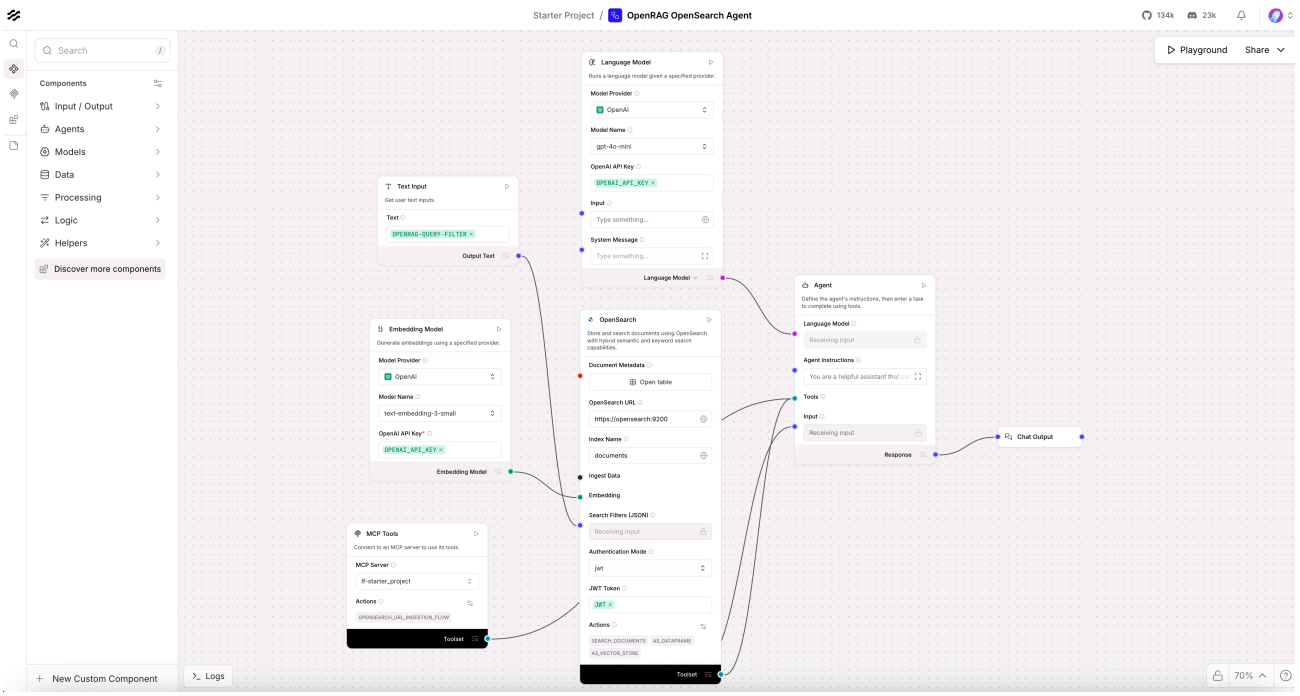
After you upload documents to your knowledge base, you can use the OpenRAG **Chat** feature to interact with your knowledge through natural language queries.

> **TIP**
>
> Try chatting, uploading documents, and modifying chat settings in the quickstart.

OpenRAG OpenSearch Agent flow

When you use the OpenRAG **Chat**, the **OpenRAG OpenSearch Agent** flow runs in the background to retrieve relevant information from your knowledge base and generate a response.

If you inspect the flow in Langflow, you'll see that it is comprised of eight components that work together to ingest chat messages, retrieve relevant information from your knowledge base, and then generate responses.



- **Chat Input** component: This component starts the flow when it receives a chat message. It is connected to the **Agent** component's **Input** port. When you use the OpenRAG **Chat**, your chat messages are passed to the **Chat Input** component, which then sends them to the **Agent** component for processing.

- **Agent** component: This component orchestrates the entire flow by processing chat messages, searching the knowledge base, and organizing the retrieved information into a cohesive response. The agent's general behavior is defined by the prompt in the **Agent Instructions** field and the model connected to the **Language Model** port. One or more specialized tools can be attached to the **Tools** port to extend the agent's capabilities. In this case, there are two tools: **MCP Tools** and **OpenSearch**.

  The **Agent** component is the star of this flow because it powers decision making, tool calling, and an LLM-driven conversational experience.

How do agents work?

Agents extend Large Language Models (LLMs) by integrating tools, which are functions that provide additional context and enable autonomous task execution. These integrations make agents more specialized and powerful than standalone LLMs.

Whereas an LLM might generate acceptable, inert responses to general queries and tasks, an agent can leverage the integrated context and tools to provide more relevant responses and even take action. For example, you might create an agent that can access your company's documentation, repositories, and other resources to help your team with tasks that require knowledge of your specific products, customers, and code.

Agents use LLMs as a reasoning engine to process input, determine which actions to take to address the query, and then generate a response. The response could be a typical text-based LLM response, or it could involve an action, like editing a file, running a script, or calling an external API.

In an agentic context, tools are functions that the agent can run to perform tasks or access external resources. A function is wrapped as a Tool object with a common interface that the agent understands. Agents become aware of tools through tool registration, which is when the agent is provided a list of available tools typically at agent initialization. The Tool object's description tells the agent what the tool can do so that it can decide whether the tool is appropriate for a given request.

- **Language Model** component: Connected to the **Agent** component's **Language Model** port, this component provides the base language model driver for the agent.

The agent cannot function without a model because the model is used for general knowledge, reasoning, and generating responses.

Different models can change the style and content of the agent's responses, and some models might be better suited for certain tasks than others. If the agent doesn't seem to be handling requests well, try changing the model to see how the responses change. For example, fast models might be good for simple queries, but they might not have the depth of reasoning for complex, multi-faceted queries.

- **MCP Tools** component: Connected to the **Agent** component's **Tools** port, this component can be used to access any Model Context Protocol (MCP) server and the MCP tools provided by that server. In this case, your OpenRAG Langflow instance's **Starter Project** is the MCP server, and the **OpenSearch URL Ingestion** flow is the MCP tool. This flow fetches content from URLs, and then stores the content in your OpenRAG OpenSearch knowledge base. By serving this flow as an MCP tool, the agent can selectively call this tool if a URL is detected in the chat input.

- **OpenSearch** component: Connected to the **Agent** component's **Tools** port, this component lets the agent search your OpenRAG OpenSearch knowledge base. The agent might not use this database for every request; the agent uses this connection only if it decides that documents in your knowledge base are relevant to your query.

- **Embedding Model** component: Connected to the **OpenSearch** component's **Embedding** port, this component generates embeddings from chat input that are used in similarity search to find content in your knowledge base that is relevant to the chat input. The agent uses this information to generate context-aware responses that are specialized for your data.

  It is critical that the embedding model used here matches the embedding model used when you upload documents to your knowledge base. Mismatched models and dimensions can degrade the quality of similarity search results causing the agent to retrieve irrelevant documents from your knowledge base.

- **Text Input** component: Connected to the **OpenSearch** component's **Search Filters** port, this component is populated with a Langflow global variable named `OPENRAG-QUERY-FILTER`. If a global or chat-level knowledge filter is set, then the variable contains the filter expression, which limits the documents that the agent can access in the knowledge base. If no knowledge filter is set, then the `OPENRAG-QUERY-`

`FILTER` variable is empty, and the agent can access all documents in the knowledge base.

- **Chat Output** component: Connected to the **Agent** component's **Output** port, this component returns the agent's generated response as a chat message.

Nudges

When you use the OpenRAG **Chat**, the **OpenRAG OpenSearch Nudges** flow runs in the background to pull additional context from your knowledge base and chat history.

Nudges appear as prompts in the chat. Click a nudge to accept it and provide the nudge's context to the OpenRAG **Chat** agent (the **OpenRAG OpenSearch Agent** flow).

Like OpenRAG's other built-in flows, you can inspect the flow in Langflow, and you can customize it if you want to change the nudge behavior.

Upload documents to the chat

When using the OpenRAG **Chat**, click + in the chat input field to upload a file to the current chat session. Files added this way are processed and made available to the agent for the current conversation only. These files aren't stored in the knowledge base permanently.

Inspect tool calls and knowledge

During the chat, you'll see information about the agent's process. For more detail, you can inspect individual tool calls. This is helpful for troubleshooting because it shows you how the agent used particular tools. For example, click **Function Call: search_documents (tool_call)** to view the log of tool calls made by the agent to the **OpenSearch** component.

If documents in your knowledge base seem to be missing or interpreted incorrectly, see Troubleshoot ingestion.

If tool calls and knowledge appear normal, but the agent's responses seem off-topic or incorrect, consider changing the agent's language model or prompt, as explained in Inspect and modify flows.

Integrate OpenRAG chat into an application

You can integrate OpenRAG flows into your applications using the Langflow API. To simplify this integration, you can get pre-configured code snippets directly from the embedded Langflow visual editor.

The following example demonstrates how to generate and use code snippets for the **OpenRAG OpenSearch Agent** flow:

1. Open the **OpenRAG OpenSearch Agent** flow in the Langflow visual editor: From the **Chat** window, click **Settings**, click **Edit in Langflow**, and then click **Proceed**.

2. Create a Langflow API key, which is a user-specific token required to send requests to the Langflow server. This key doesn't grant access to OpenRAG.

    i. In the Langflow visual editor, click your user icon in the header, and then select **Settings**.
    ii. Click **Langflow API Keys**, and then click **Add New**.
    iii. Name your key, and then click **Create API Key**.
    iv. Copy the API key and store it securely.
    v. Exit the Langflow **Settings** page to return to the visual editor.
3. Click **Share**, and then select **API access** to get pregenerated code snippets that call the Langflow API and run the flow.

    These code snippets construct API requests with your Langflow server URL (`LANGFLOW_SERVER_ADDRESS`), the flow to run (`FLOW_ID`), required headers (`LANGFLOW_API_KEY`, `Content-Type`), and a payload containing the required inputs to run the flow, including a default chat input message.

    In production, you would modify the inputs to suit your application logic. For example, you could replace the default chat input message with dynamic user input.

Python

```
import requests
import os
import uuid
api_key = 'LANGFLOW_API_KEY'
url = "http://LANGFLOW_SERVER_ADDRESS/api/v1/run/FLOW_ID"  #
The complete API endpoint URL for this flow
# Request payload configuration
```

```
payload = {
    "output_type": "chat",
    "input_type": "chat",
    "input_value": "hello world!"
}
payload["session_id"] = str(uuid.uuid4())
headers = {"x-api-key": api_key}
try:
    # Send API request
    response = requests.request("POST", url, json=payload,
headers=headers)
    response.raise_for_status()  # Raise exception for bad
status codes
    # Print response
    print(response.text)
except requests.exceptions.RequestException as e:
    print(f"Error making API request: {e}")
except ValueError as e:
    print(f"Error parsing response: {e}")
```

TypeScript

```
const crypto = require('crypto');
const apiKey = 'LANGFLOW_API_KEY';
const payload = {
    "output_type": "chat",
    "input_type": "chat",
    "input_value": "hello world!"
};
payload.session_id = crypto.randomUUID();
const options = {
    method: 'POST',
    headers: {
        'Content-Type': 'application/json',
        "x-api-key": apiKey
    },
    body: JSON.stringify(payload)
};
fetch('http://LANGFLOW_SERVER_ADDRESS/api/v1/run/FLOW_ID',
options)
    .then(response => response.json())
```

```
    .then(response => console.warn(response))
    .catch(err => console.error(err));
```

curl

```
curl --request POST \
  --url 'http://LANGFLOW_SERVER_ADDRESS/api/v1/run/FLOW_ID?
stream=false' \
  --header 'Content-Type: application/json' \
  --header "x-api-key: LANGFLOW_API_KEY" \
  --data '{
    "output_type": "chat",
    "input_type": "chat",
    "input_value": "hello world!"
  }'
```

4. Copy your preferred snippet, and then run it:

    - **Python**: Paste the snippet into a `.py` file, save it, and then run it with `python filename.py`.
    - **TypeScript**: Paste the snippet into a `.ts` file, save it, and then run it with `ts-node filename.ts`.
    - **curl**: Paste and run snippet directly in your terminal.

If the request is successful, the response includes many details about the flow run, including the session ID, inputs, outputs, components, durations, and more.

In production, you won't pass the raw response to the user in its entirety. Instead, you extract and reformat relevant fields for different use cases, as demonstrated in the Langflow quickstart. For example, you could pass the chat output text to a front-end user-facing application, and store specific fields in logs and backend data stores for monitoring, chat history, or analytics. You could also pass the output from one flow as input to another flow.

Environment variables

OpenRAG recognizes environment variables from the following sources:

- Environment variables: Values set in the `.env` file.
- Langflow runtime overrides: Langflow components can set environment variables at runtime.
- Default or fallback values: These values are default or fallback values if OpenRAG doesn't find a value.

Configure environment variables

Environment variables are set in a `.env` file in the root of your OpenRAG project directory.

For an example `.env` file, see `.env.example` in the OpenRAG repository.

The Docker Compose files are populated with values from your `.env`, so you don't need to edit the Docker Compose files manually.

Environment variables always take precedence over other variables.

Set environment variables

After you start OpenRAG, you must stop and restart OpenRAG containers to apply any changes you make to the `.env` file.

To set mutable environment variables, do the following:

1. Stop OpenRAG with the TUI or Docker Compose.

2. Set the values in the `.env` file:

```
LOG_LEVEL=DEBUG
LOG_FORMAT=json
SERVICE_NAME=openrag-dev
```

3. Start OpenRAG with the TUI or Docker Compose.

Certain environment variables that you set during application onboarding, such as provider API keys and provider endpoints, require resetting the containers after modifying the `.env` file.

To change immutable variables with TUI-managed containers, you must reinstall OpenRAG and either delete or modify the `.env` file before you repeat the setup and onboarding process in the TUI.

To change immutable variables with self-managed containers, do the following:

1. Stop OpenRAG with Docker Compose.
2. Remove the containers:

   ```
   docker-compose down
   ```

3. Update the values in your `.env` file.
4. Start OpenRAG with Docker Compose:

   ```
   docker-compose up -d
   ```

5. Repeat application onboarding. The values in your `.env` file are automatically populated.

Supported environment variables

All OpenRAG configuration can be controlled through environment variables.

AI provider settings

Configure which models and providers OpenRAG uses to generate text and embeddings. These are initially set during application onboarding. Some values are immutable and can only be changed by recreating the OpenRAG containers, as explained in Set environment variables.

Variable	Default	Description
`EMBEDDING_MODEL`	`text-embedding-3-small`	Embedding model for generating vector embeddings for documents in the knowledge base and similarity search queries. Can be changed after application onboarding. Accepts one or more models.
`LLM_MODEL`	`gpt-4o-mini`	Language model for language processing and text generation in the **Chat** feature.
`MODEL_PROVIDER`	`openai`	Model provider, such as OpenAI or IBM watsonx.ai.
`OPENAI_API_KEY`	Not set	Optional OpenAI API key for the default model. For other providers, use `PROVIDER_API_KEY`.
`PROVIDER_API_KEY`	Not set	API key for the model provider.
`PROVIDER_ENDPOINT`	Not set	Custom provider endpoint for the IBM and Ollama model providers. Leave unset for other model providers.
`PROVIDER_PROJECT_ID`	Not set	Project ID for the IBM watsonx.ai model provider only. Leave unset for other model providers.

Document processing

Control how OpenRAG processes and ingests documents into your knowledge base.

Variable	Default	Description
`CHUNK_OVERLAP`	`200`	Overlap between chunks.
`CHUNK_SIZE`	`1000`	Text chunk size for document processing.
`DISABLE_INGEST_WITH_LANGFLOW`	`false`	Disable Langflow ingestion pipeline.
`DOCLING_OCR_ENGINE`	Set by OS	OCR engine for document processing. For macOS,

Variable	Default	Description
		`ocrmac`. For any other OS, `easyocr`.
`OCR_ENABLED`	`false`	Enable OCR for image processing.
`OPENRAG_DOCUMENTS_PATHS`	`./openrag-documents`	Document paths for ingestion.
`PICTURE_DESCRIPTIONS_ENABLED`	`false`	Enable picture descriptions.

Langflow settings

Configure Langflow authentication.

Variable	Default	Description
`LANGFLOW_AUTO_LOGIN`	`False`	Enable auto-login for Langflow.
`LANGFLOW_CHAT_FLOW_ID`	Built-in flow ID	This value is automatically set to the ID of the chat flow. The default value is found in `.env.example` Only change this value if you explicitly don't want to use this built-in flow.
`LANGFLOW_ENABLE_SUPERUSER_CLI`	`False`	Enable superuser privileges for Langflow CLI commands.
`LANGFLOW_INGEST_FLOW_ID`	Built-in flow ID	This value is automatically set to the ID of the ingestion flow. The

Variable	Default	Description
		default value is found in `.env.example` Only change this value if you explicitly don't want to use this built-in flow.
`LANGFLOW_KEY`	Automatically generated	Explicit Langflow API key.
`LANGFLOW_NEW_USER_IS_ACTIVE`	`False`	Whether new Langflow users are active by default.
`LANGFLOW_PUBLIC_URL`	`http://localhost:7860`	Public URL for the Langflow instance.
`LANGFLOW_SECRET_KEY`	Not set	Secret key for Langflow internal operations.
`LANGFLOW_SUPERUSER`	None, must be explicitly set	Langflow admin username. Required.
`LANGFLOW_SUPERUSER_PASSWORD`	None, must be explicitly set	Langflow admin password. Required.
`LANGFLOW_URL`	`http://localhost:7860`	URL for the Langflow instance.
`NUDGES_FLOW_ID`	Built-in flow ID	This value is automatically set to the ID of the nudges flow The default value is found in `.env.example` Only change this value if you

Variable	Default	Description
		explicitly don't want to use this built-in flow.
`SYSTEM_PROMPT`	`You are a helpful AI assistant with access to a knowledge base. Answer questions based on the provided context.`	System prompt instructions for the agent driving the **Chat** flow.

OAuth provider settings

Configure OAuth providers and external service integrations.

Variable	Default	Description
`AWS_ACCESS_KEY_ID` / `AWS_SECRET_ACCESS_KEY`	-	AWS integrations.
`GOOGLE_OAUTH_CLIENT_ID` / `GOOGLE_OAUTH_CLIENT_SECRET`	-	Google OAuth authentication.
`MICROSOFT_GRAPH_OAUTH_CLIENT_ID` / `MICROSOFT_GRAPH_OAUTH_CLIENT_SECRET`	-	Microsoft OAuth.
`WEBHOOK_BASE_URL`	-	Base URL for webhook endpoints.

OpenSearch settings

Configure OpenSearch database authentication.

Variable	Default	Description
`OPENSEARCH_HOST`	`localhost`	OpenSearch host.
`OPENSEARCH_PASSWORD`	-	Password for OpenSearch admin user. Required.
`OPENSEARCH_PORT`	`9200`	OpenSearch port.

Variable	Default	Description
`OPENSEARCH_USERNAME`	`admin`	OpenSearch username.

System settings

Configure general system components, session management, and logging.

Variable	Default	Description
`LANGFLOW_KEY_RETRIES`	`15`	Number of retries for Langflow key generation.
`LANGFLOW_KEY_RETRY_DELAY`	`2.0`	Delay between retries in seconds.
`LANGFLOW_VERSION`	`OPENRAG_VERSION`	Langflow Docker image version. By default, OpenRAG uses the `OPENRAG_VERSION` for the Langflow Docker image version.
`LOG_FORMAT`	Disabled	Set to `json` to enabled JSON-formatted log output.
`LOG_LEVEL`	`INFO`	Logging level (DEBUG, INFO, WARNING, ERROR).
`MAX_WORKERS`	`1`	Maximum number of workers for document processing.
`OPENRAG_VERSION`	`latest`	The version of the OpenRAG Docker images to run. For more information, see Upgrade OpenRAG
`SERVICE_NAME`	`openrag`	Service name for logging.
`SESSION_SECRET`	Automatically generated	Session management.

Langflow runtime overrides

You can modify flow settings at runtime without permanently changing the flow's configuration.

Runtime overrides are implemented through *tweaks*, which are one-time parameter modifications that are passed to specific Langflow components during flow execution.

For more information on tweaks, see the Langflow documentation on Input schema (tweaks).

Default values and fallbacks

If a variable isn't set by environment variables or a configuration file, OpenRAG can use a default value if one is defined in the codebase. Default values can be found in the OpenRAG repository:

- OpenRAG configuration: `config_manager.py`
- System configuration: `settings.py`
- Logging configuration: `logging_config.py`

Troubleshoot OpenRAG

This page provides troubleshooting advice for issues you might encounter when using OpenRAG or contributing to OpenRAG.

OpenSearch fails to start

Check that `OPENSEARCH_PASSWORD` set in Environment variables meets requirements. The password must contain at least 8 characters, and must contain at least one uppercase letter, one lowercase letter, one digit, and one special character that is strong.

OpenRAG fails to start from the TUI with operation not supported

This error occurs when starting OpenRAG with the TUI in WSL (Windows Subsystem for Linux).

The error occurs because OpenRAG is running within a WSL environment, so `webbrowser.open()` can't launch a browser automatically.

To access the OpenRAG application, open a web browser and enter `http://localhost:3000` in the address bar.

OpenRAG installation fails with unable to get local issuer certificate

If you are installing OpenRAG on macOS, and the installation fails with `unable to get local issuer certificate`, run the following command, and then retry the installation:

```
open "/Applications/Python VERSION/Install Certificates.command"
```

Replace `VERSION` with your installed Python version, such as `3.13`.

Langflow connection issues

Verify the `LANGFLOW_SUPERUSER` credentials set in Environment variables are correct.

Container out of memory errors

Increase Docker memory allocation or use docker-compose-cpu.yml to deploy OpenRAG.

Memory issue with Podman on macOS

If you're using Podman on macOS, you might need to increase VM memory on your Podman machine. This example increases the machine size to 8 GB of RAM, which should be sufficient to run OpenRAG.

```
podman machine stop
podman machine rm
podman machine init --memory 8192   # 8 GB example
podman machine start
```

Port conflicts

Ensure ports 3000, 7860, 8000, 9200, 5601 are available.

OCR ingestion fails (easyocr not installed)

If Docling ingestion fails with an OCR-related error and mentions `easyocr` is missing, this is likely due to a stale `uv` cache.

`easyocr` is already included as a dependency in OpenRAG's `pyproject.toml`. Project-managed installations using `uv sync` and `uv run` always sync dependencies directly from your `pyproject.toml`, so they should have `easyocr` installed.

If you're running OpenRAG with `uvx openrag`, `uvx` creates a cached, ephemeral environment that doesn't modify your project. This cache can become stale.

On macOS, this cache directory is typically a user cache directory such as `/Users/USER_NAME/.cache/uv`.

1. To clear the uv cache, run:

   ```
   uv cache clean
   ```

2. Start OpenRAG:

```
uvx openrag
```

If you don't need OCR, you can disable OCR-based processing in your ingestion settings to avoid requiring `easyocr`.

Upgrade fails due to Langflow container already exists

If you encounter a `langflow container already exists` error when upgrading OpenRAG, this typically means you upgraded OpenRAG with `uv`, but you didn't remove or upgrade containers from a previous installation.

To resolve this issue, do the following:

First, try removing only the Langflow container, and then retry the upgrade in the OpenRAG TUI by clicking **Status** and then **Upgrade**.

Podman

1. Stop the Langflow container:

```
podman stop langflow
```

2. Remove the Langflow container:

```
podman rm langflow --force
```

Docker

1. Stop the Langflow container:

```
docker stop langflow
```

2. Remove the Langflow container:

```
docker rm langflow --force
```

If reinstalling the Langflow container doesn't resolve the issue, you must reset to a fresh installation by removing all OpenRAG containers and data. Then, you can retry the upgrade.

> **WARNING**
> This is a destructive operation that destroys your OpenRAG containers and their contents. However, your `.env` file (configuration settings) and `./opensearch-data` (OpenSearch knowledge base) are preserved.

To reset your installation, stop your containers, and then completely remove them. After removing the containers, retry the upgrade in the OpenRAG TUI by clicking **Status** and then **Upgrade**.

Podman

1. Stop all running containers:

   ```
   podman stop --all
   ```

2. Remove all containers, including stopped containers:

   ```
   podman rm --all --force
   ```

3. Remove all images:

   ```
   podman rmi --all --force
   ```

4. Remove all volumes:

   ```
   podman volume prune --force
   ```

5. Remove all networks except the default network:

```
podman network prune --force
```

6. Clean up any leftover data:

```
podman system prune --all --force --volumes
```

Docker

1. Stop all running containers:

```
docker stop $(docker ps -q)
```

2. Remove all containers, including stopped containers:

```
docker rm --force $(docker ps -aq)
```

3. Remove all images:

```
docker rmi --force $(docker images -q)
```

4. Remove all volumes:

```
docker volume prune --force
```

5. Remove all networks except the default network:

```
docker network prune --force
```

6. Clean up any leftover data:

```
docker system prune --all --force --volumes
```

Reinstalling OpenRAG doesn't reset onboarding

If you reinstall OpenRAG, you can restore your installation to it's original, default state by resetting the containers *and* deleting the `.env` file.

When you start OpenRAG after doing this, you should be prompted to go through the initial setup and onboarding process again.

Due to a known issue, the onboarding process might not reset when you reinstall OpenRAG. If this occurs, install OpenRAG in a new Python project directory (with `uv init` and `uv add openrag`).

Document ingestion or similarity search issues

See Troubleshoot ingestion.