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

- For Microsoft Windows, you must use the Windows Subsystem for Linux (WSL). See Install OpenRAG on Windows before proceeding.
- Get an OpenAI API key. This quickstart uses OpenAI for simplicity. For other providers, see the other installation methods.
- Install Python version 3.13 or later.

Install OpenRAG

For this quickstart, install OpenRAG with the automatic installer script and basic setup. The script installs OpenRAG dependencies, including Docker or Podman, and then it installs and runs OpenRAG with `uvx`.

1. Create a directory for your OpenRAG installation, and then change to that directory:

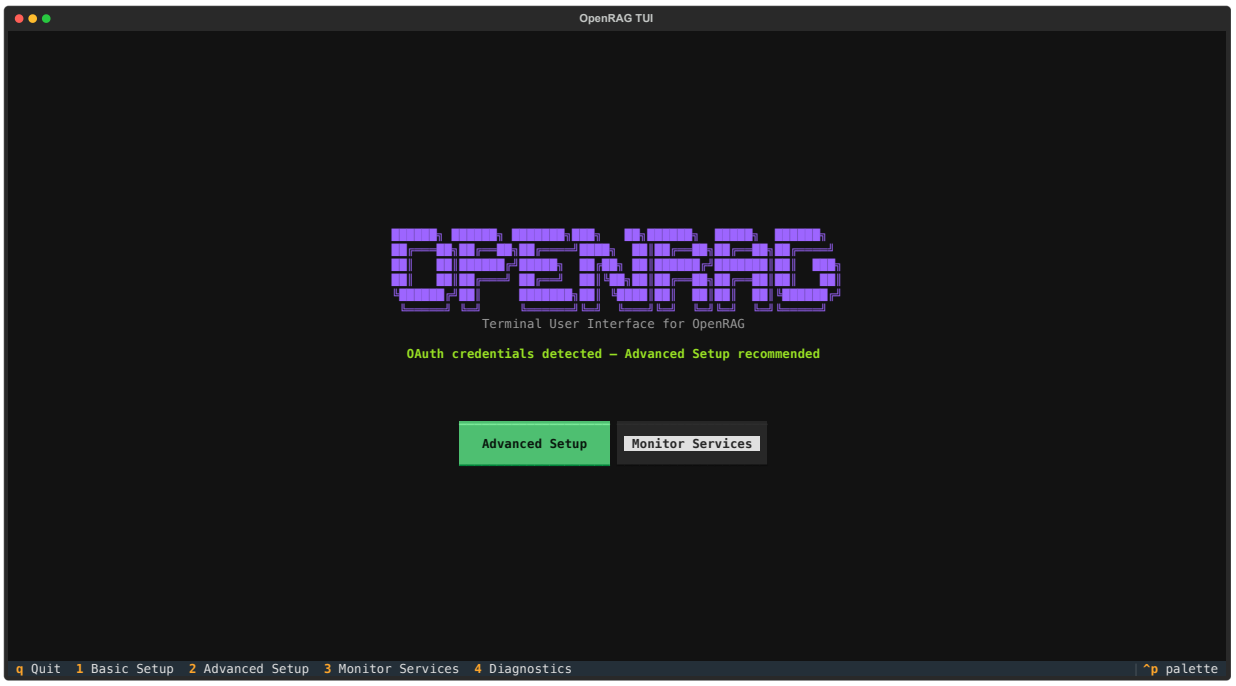
   ```
   mkdir openrag-workspace
   cd openrag-workspace
   ```

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

   ```
   bash run_openrag_with_prereqs.sh
   ```

   Wait while the installer script prepares your environment and installs OpenRAG. You might be prompted to install certain dependencies if they aren't already present in your environment.

   The entire process can take a few minutes. Once the environment is ready, the OpenRAG Terminal User Interface (TUI) starts.



3. In the TUI, click **Basic Setup**.

4. Click **Generate Passwords** to create administrator passwords for your OpenRAG OpenSearch and Langflow services.

5. Leave the **OpenAI API key** field empty.

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

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

   This process can take some time while OpenRAG pulls and runs the container images. If all services start successfully, the TUI prints a confirmation message:

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

   Your OpenRAG configuration is stored in a `.env` file that is created automatically in the directory where you ran the installer script. Container definitions are stored in the `docker-compose` files in the same directory.

7. Under **Native Services**, click **Start** to start the Docling service.

8. From the TUI main menu, click **Open App** to launch the OpenRAG application and start the application onboarding process.

9. For this quickstart, select the **OpenAI** model provider, enter your OpenAI API key, and then click **Complete**. Use the default settings for all other model options.

10. Click through the overview slides for a brief introduction to OpenRAG, or click **Skip overview**. You can complete this quickstart without going through the overview. The overview demonstrates some basic functionality that is covered in the next section and in other parts of the OpenRAG documentation.

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

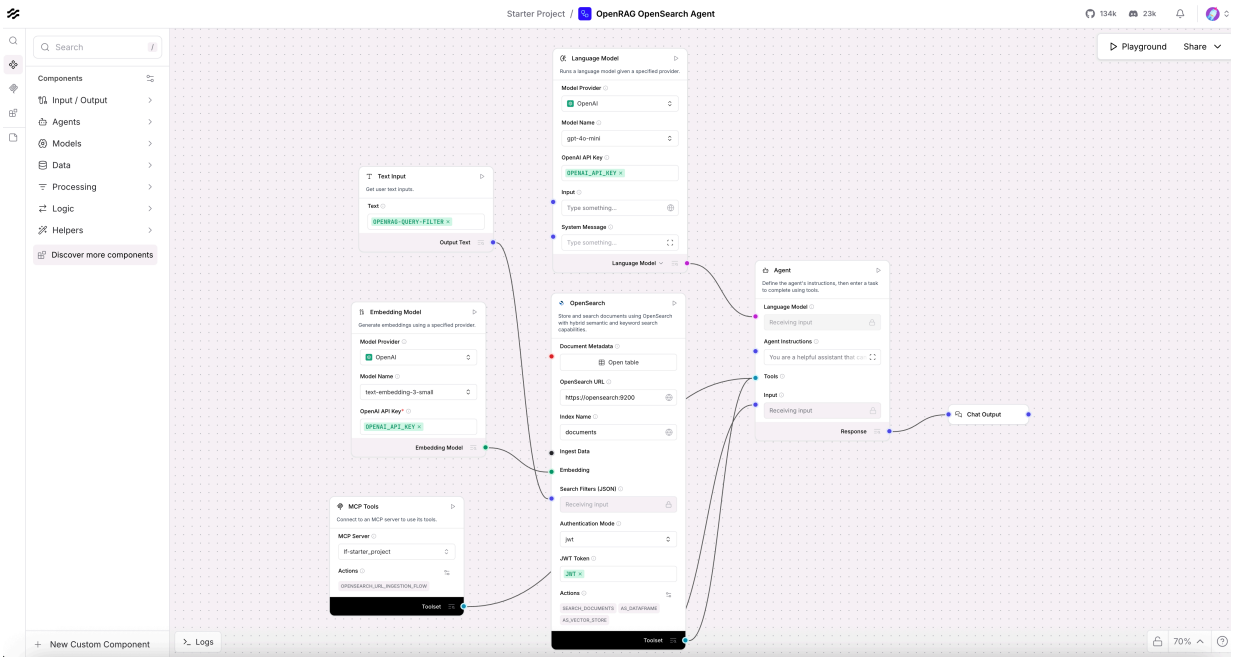
1. To change the knowledge ingestion settings, agent behavior, or language model, click **Settings**.

   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. Optional: If you don't want to use the Langflow API key that is generated automatically when you install OpenRAG, you can create a Langflow API key. This key doesn't grant access to OpenRAG; it is only for authenticating with the Langflow API.

    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 `uvx` and a minimal setup to demonstrate OpenRAG's core functionality. It is recommended that you reinstall OpenRAG with your preferred configuration and installation method.

- **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.

Select an installation method

The OpenRAG architecture is lightweight and container-based with a central OpenRAG backend that orchestrates the various services and external connectors. Depending on your use case, OpenRAG can assist with service management, or you can manage the services yourself.

Select the installation method that best fits your needs:

- **Use the Terminal User Interface (TUI) to manage services**: For guided configuration and simplified service management, install OpenRAG with TUI-managed services.

    - **Automatic installer script**: Run one script to install the required dependencies and OpenRAG.
    - `uv`: Install OpenRAG as a dependency of a new or existing Python project.
    - `uvx`: Install OpenRAG without creating a project or modifying your project's dependencies.
- **Install OpenRAG on Microsoft Windows**: On Windows machines, you must install OpenRAG within the Windows Subsystem for Linux (WSL).

    OpenRAG doesn't support nested virtualization; don't run OpenRAG on a WSL distribution that is inside a Windows VM.

- **Manage your own services**: You can use Docker or Podman to deploy self-managed OpenRAG services.

The first time you start OpenRAG, you must complete application onboarding. This is required for all installation methods because it prepares the minimum required configuration for OpenRAG to run. For TUI-managed services, you must also complete initial setup before you start the OpenRAG services. For more information, see the instructions for your preferred installation method.

Your OpenRAG configuration is stored in a `.env` file in the OpenRAG installation directory. When using TUI-managed services, the TUI prompts you for any missing values during setup and onboarding, and any values detected in a preexisting `.env` file are automatically populated. When using self-managed services, you must predefine these values in a `.env` file, as you would for any Docker or Podman deployment. For

more information, see the instructions for your preferred installation method and Environment variables.

Install OpenRAG with the automatic installer script

> **TIP**
> For a fully guided installation and preview of OpenRAG's core features, try the quickstart.

For guided configuration and simplified service management, install OpenRAG with services managed by the Terminal User Interface (TUI).

The installer script installs `uv`, Docker or Podman, Docker Compose, and OpenRAG.

This installation method is best for testing OpenRAG by running it outside of a Python project. For other installation methods, see Select an installation method.

Prerequisites

- For Microsoft Windows, you must use the Windows Subsystem for Linux (WSL). See Install OpenRAG on Windows before proceeding.
- Gather the credentials and connection details for your preferred model providers. You must have access to at least one language model and one embedding model. If a provider offers both types, you can use the same provider for both models. If a provider offers only one type, you must select two providers.
    - **OpenAI**: Create an OpenAI API key.
    - **Anthropic**: Create an Anthropic API key. Anthropic provides language models only; you must select an additional provider for embeddings.
    - **IBM watsonx.ai**: Get your watsonx.ai API endpoint, IBM project ID, and IBM API key from your watsonx deployment.
    - **Ollama**: Deploy an Ollama instance and models locally, in the cloud, or on a remote server, and then get your Ollama server's base URL and the names of the models that you want to use.
- 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 Python version 3.13 or later.

Run the installer script

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

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

2. Get and run the installer script:

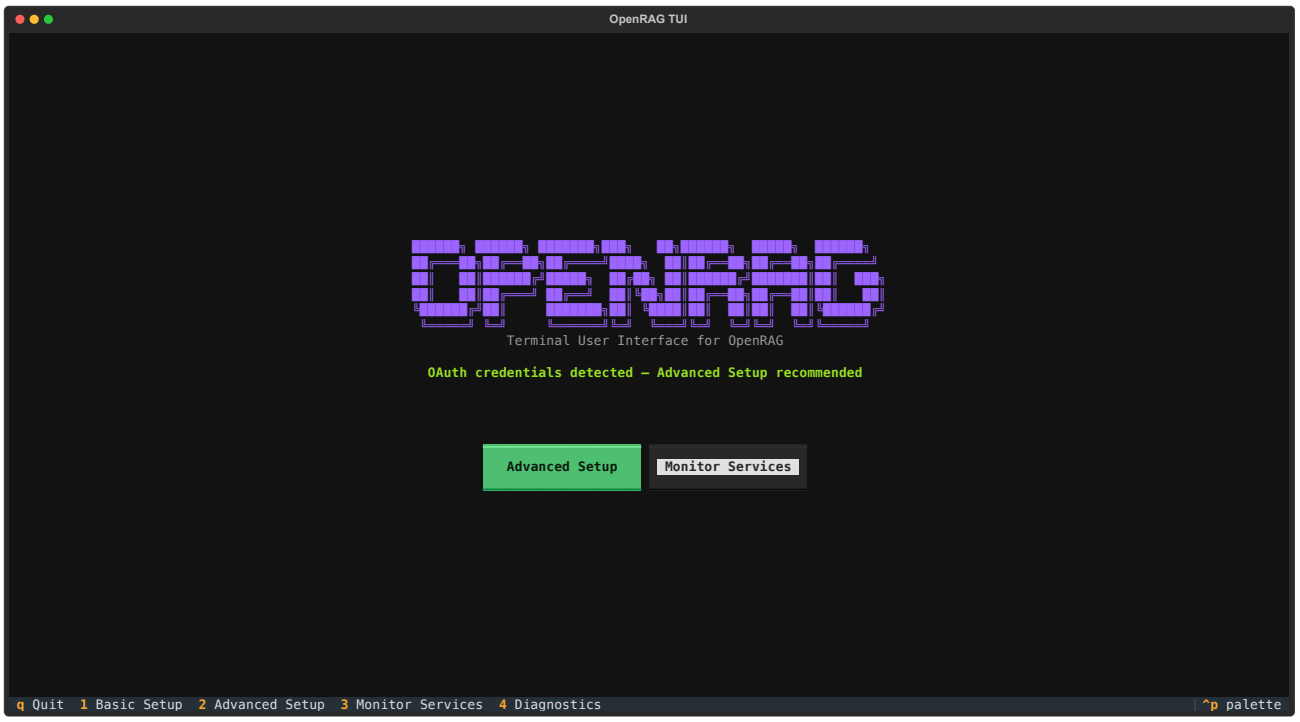
   ```
   curl -fsSL
   https://docs.openr.ag/files/run_openrag_with_prereqs.sh | bash
   ```

   > **TIP**
   >
   > You can also manually download the OpenRAG install script, move it to your OpenRAG directory, and then run it:
   >
   > ```
   > bash run_openrag_with_prereqs.sh
   > ```

   The installer script installs OpenRAG with `uvx` in the directory where you run the script.

3. Wait while the installer script prepares your environment and installs OpenRAG. You might be prompted to install certain dependencies if they aren't already present in your environment.

The entire process can take a few minutes. Once the environment is ready, the OpenRAG Terminal User Interface (TUI) starts.



Because the installer script uses `uvx`, it creates a cached, ephemeral environment in your local `uv` cache, and your OpenRAG configuration files and data are stored separately from the `uv` cache. Clearing the cache doesn't delete your entire OpenRAG installation, only the temporary TUI environment. After clearing the cache, run `uvx openrag` to access the TUI and continue with your preserved configuration and data.

If you encounter errors during installation, see Troubleshoot OpenRAG.

Set up OpenRAG with the TUI

When you install OpenRAG with the installer script, you manage the OpenRAG services with the Terminal User Interface (TUI). The TUI guides you through the initial configuration process before you start the OpenRAG services.

Your OpenRAG configuration is stored in a `.env` file that is created automatically in the OpenRAG installation directory. If OpenRAG detects an existing `.env` file, the TUI automatically populates those values during setup and onboarding.

Container definitions are stored in the `docker-compose` files in the OpenRAG installation directory.

Because the installer script uses `uvx`, the OpenRAG `.env` and `docker-compose` files are stored in the directory where you ran the installer script.

You can use either **Basic Setup** or **Advanced Setup** to configure OpenRAG. This choice determines how OpenRAG authenticates with OpenSearch and controls access to documents.

> **INFO**
>
> You must use **Advanced Setup** if you want to use OAuth connectors to upload documents from cloud storage.

If OpenRAG detects OAuth credentials during setup, it recommends **Advanced Setup** in the TUI.

Basic setup

1. In the TUI, click **Basic Setup** or press `1`.
2. Enter administrator passwords for the OpenRAG OpenSearch and Langflow services, or click **Generate Passwords** to generate passwords automatically.

   The OpenSearch password is required.

   The Langflow password is recommended but optional. If the Langflow password is empty, the Langflow server starts without authentication enabled. For more information, see Langflow settings.
3. Optional: Enter your OpenAI API key, or leave this field empty if you want to configure model provider credentials later during application onboarding.
4. Click **Save Configuration**.

   Your passwords and API key, if provided, are stored in the `.env` file in your OpenRAG installation directory. If you modified any credentials that were pulled from an existing `.env` file, those values are updated in the `.env` file.
5. Click **Start All Services** to start the OpenRAG services that run in containers.

   This process can take some time while OpenRAG pulls and runs the container images. If all services start successfully, the TUI prints a confirmation message:

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

6. Under **Native Services**, click **Start** to start the Docling service.

7. Launch the OpenRAG application:

    - From the TUI main menu, click **Open App**.
    - In your browser, navigate to `localhost:3000`.

8. Continue with application onboarding.

Advanced setup

1. In the TUI, click **Advanced Setup** or press `2`.

2. Enter administrator passwords for the OpenRAG OpenSearch and Langflow services, or click **Generate Passwords** to generate passwords automatically.

    The OpenSearch password is required.

    The Langflow password is recommended but optional. If the Langflow password is empty, the Langflow server starts without authentication enabled. For more information, see Langflow settings.

3. Optional: Enter your OpenAI API key, or leave this field empty if you want to configure model provider credentials later during application onboarding.

4. 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**.

   Your passwords, API key, and OAuth credentials, if provided, are stored in the `.env` file in your OpenRAG installation directory. If you modified any credentials that were pulled from an existing `.env` file, those values are updated in the `.env` file.

7. Click **Start All Services** to start the OpenRAG services that run in containers.

   This process can take some time while OpenRAG pulls and runs the container images. If all services start successfully, the TUI prints a confirmation message:

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

8. Under **Native Services**, click **Start** to start the Docling service.

9. Launch the OpenRAG application:

   - From the TUI main menu, click **Open App**.
   - 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. If required, you can edit the following additional environment variables. Only change these variables if your OpenRAG deployment has a non-default network configuration, such as 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 for the following OpenRAG OAuth connector endpoints:

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

12. Continue with application onboarding.

Application onboarding

The first time you start the OpenRAG application, you must complete application onboarding to select language and embedding models that are essential for OpenRAG features like the **Chat**.

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 embedding 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 the embedding model.

1. Enter your Anthropic API key, or enable **Get API key from environment variable** to pull the key from your OpenRAG `.env` file.

    If you haven't set `ANTHROPIC_API_KEY` in your `.env` file, you must enter the key manually.

2. Under **Advanced settings**, select the language model that you want to use.

3. Click **Complete**.

4. Select a provider for embeddings, provide the required information, and then select the embedding model you want to use. For information about another provider's credentials and settings, see the instructions for that provider.

5. Click **Complete**.

   After you configure the embedding model, OpenRAG uses your credentials and models to ingest some initial documents. This tests the connection, and it allows you to ask OpenRAG about itself in the **Chat**. If there is a problem with the model configuration, an error occurs and you are redirected back to application onboarding. Verify that the credential is valid and has access to the selected model, and then click **Complete** to retry ingestion.

6. Continue through the overview slides for a brief introduction to OpenRAG, or click **Skip overview**. The overview demonstrates some basic functionality that is covered in the quickstart and in other parts of the OpenRAG documentation.

IBM watsonx.ai

1. Use the values from your IBM watsonx deployment for the **watsonx.ai API Endpoint**, **IBM Project ID**, and **IBM API key** fields.

2. Under **Advanced settings**, select the language model that you want to use.

3. Click **Complete**.

4. Select a provider for embeddings, provide the required information, and then select the embedding model you want to use. For information about another provider's credentials and settings, see the instructions for that provider.

5. Click **Complete**.

   After you configure the embedding model, OpenRAG uses your credentials and models to ingest some initial documents. This tests the connection, and it allows you to ask OpenRAG about itself in the **Chat**. If there is a problem with the model configuration, an error occurs and you are redirected back to application

onboarding. Verify that the credentials are valid and have access to the selected model, and then click **Complete** to retry ingestion.

6. Continue through the overview slides for a brief introduction to OpenRAG, or click → **Skip overview**. The overview demonstrates some basic functionality that is covered in the quickstart and in other parts of the OpenRAG documentation.

Ollama

> **INFO**
> Ollama isn't installed with OpenRAG. You must install it separately if you want to use Ollama as a model provider.

Using Ollama as your language and embedding model provider offers greater flexibility and configuration options for hosting models, but it can be advanced for new users. The recommendations given here are a reasonable starting point for users with at least one GPU and experience running LLMs locally.

The OpenRAG team recommends the OpenAI `gpt-oss:20b` lanuage model and the `nomic-embed-text` embedding model. However, `gpt-oss:20b` uses 16GB of RAM, so consider using Ollama Cloud or running Ollama on a remote machine.

1. Install Ollama locally or on a remote server or run models in Ollama Cloud.

   If you are running a remote server, it must be accessible from your OpenRAG deployment.

2. In OpenRAG onboarding, connect to your Ollama server:

   - **Local Ollama server**: Enter your Ollama server's base URL and port. The default Ollama server address is `http://localhost:11434`.
   - **Ollama Cloud**: Because Ollama Cloud models run at the same address as a local Ollama server and automatically offload to Ollama's cloud service, you can use the same base URL and port as you would for a local Ollama server. The default address is `http://localhost:11434`.
   - **Remote server**: Enter your remote Ollama server's base URL and port, such as `http://your-remote-server:11434`.

If the connection succeeds, OpenRAG populates the model lists with the server's available models.

3. Select the model that your Ollama server is running.

   Language model and embedding model selections are independent. You can use the same or different servers for each model.

   To use different providers for each model, you must configure both providers, and select the relevant model for each provider.

4. Click **Complete**.

   After you configure the embedding model, OpenRAG uses the address and models to ingest some initial documents. This tests the connection, and it allows you to ask OpenRAG about itself in the **Chat**. If there is a problem with the model configuration, an error occurs and you are redirected back to application onboarding. Verify that the server address is valid, and that the selected model is running on the server. Then, click **Complete** to retry ingestion.

5. Continue through the overview slides for a brief introduction to OpenRAG, or click → **Skip overview**. The overview demonstrates some basic functionality that is covered in the quickstart and in other parts of the OpenRAG documentation.

OpenAI (default)

1. Enter your OpenAI API key, or enable **Get API key from environment variable** to pull the key from your OpenRAG `.env` file.

   If you entered an OpenAI API key during setup, enable **Get API key from environment variable**.

2. Under **Advanced settings**, select the language model that you want to use.

3. Click **Complete**.

4. Select a provider for embeddings, provide the required information, and then select the embedding model you want to use. For information about another provider's credentials and settings, see the instructions for that provider.

5. Click **Complete**.

   After you configure the embedding model, OpenRAG uses your credentials and models to ingest some initial documents. This tests the connection, and it allows you to ask OpenRAG about itself in the **Chat**. If there is a problem with the model configuration, an error occurs and you are redirected back to application onboarding. Verify that the credential is valid and has access to the selected model, and then click **Complete** to retry ingestion.

6. Continue through the overview slides for a brief introduction to OpenRAG, or click **Skip overview** →. The overview demonstrates some basic functionality that is covered in the quickstart and in other parts of the OpenRAG documentation.

Next steps

- Try some of OpenRAG's core features in the quickstart.
- Learn how to manage OpenRAG services.
- Upload documents, and then use the **Chat** to explore your data.

Install OpenRAG in a Python project with uv

For guided configuration and simplified service management, install OpenRAG with services managed by the Terminal User Interface (TUI).

You can use `uv` to install OpenRAG as a managed or unmanaged dependency in a new or existing Python project.

For other installation methods, see Select an installation method.

Prerequisites

- For Microsoft Windows, you must use the Windows Subsystem for Linux (WSL). See Install OpenRAG on Windows before proceeding.
- Gather the credentials and connection details for your preferred model providers. You must have access to at least one language model and one embedding model. If a provider offers both types, you can use the same provider for both models. If a provider offers only one type, you must select two providers.
    - **OpenAI**: Create an OpenAI API key.
    - **Anthropic**: Create an Anthropic API key. Anthropic provides language models only; you must select an additional provider for embeddings.
    - **IBM watsonx.ai**: Get your watsonx.ai API endpoint, IBM project ID, and IBM API key from your watsonx deployment.
    - **Ollama**: Deploy an Ollama instance and models locally, in the cloud, or on a remote server, and then get your Ollama server's base URL and the names of the models that you want to use.
- 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 Python version 3.13 or later.
- Install uv.
- Install Podman (recommended) or Docker.

- Install `podman-compose` or Docker Compose. To use Docker Compose with Podman, you must alias Docker Compose commands to Podman commands.

Install and start OpenRAG with uv

There are two ways to install OpenRAG with `uv`:

- **`uv add`** (Recommended): Install OpenRAG as a managed dependency in a new or existing `uv` Python project. This is recommended because it adds OpenRAG to your `pyproject.toml` and lockfile for better management of dependencies and the virtual environment.

- **`uv pip install`**: Use the `uv pip` interface to install OpenRAG into an existing Python project that uses `pip`, `pip-tools`, and `virtualenv` commands.

If you encounter errors during installation, see Troubleshoot OpenRAG.

uv add

1. Create a new `uv`-managed Python project:

   ```
   uv init PROJECT_NAME
   ```

2. Change into your new project directory:

   ```
   cd PROJECT_NAME
   ```

   Because `uv` manages the virtual environment for you, you won't see a `(venv)` prompt. `uv` commands automatically use the project's virtual environment.

3. Add OpenRAG to your project:

   - Add the latest version:

     ```
     uv add openrag
     ```

   - Add a specific version:

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

- Add a local wheel:

```
uv add path/to/openrag-VERSION-py3-none-any.whl
```

For more options, see Managing dependencies with `uv`.

4. Start the OpenRAG TUI:

```
uv run openrag
```

uv pip install

1. Activate your virtual environment.

2. Install the OpenRAG Python package:

```
uv pip install openrag
```

3. Start the OpenRAG TUI:

```
uv run openrag
```

Set up OpenRAG with the TUI

When you install OpenRAG with `uv`, you manage the OpenRAG services with the Terminal User Interface (TUI). The TUI guides you through the initial configuration process before you start the OpenRAG services.

Your OpenRAG configuration is stored in a `.env` file that is created automatically in the Python project where you installed OpenRAG. If OpenRAG detects an existing `.env` file, the TUI automatically populates those values during setup and onboarding. Container definitions are stored in the `docker-compose` files in the same directory.

You can use either **Basic Setup** or **Advanced Setup** to configure OpenRAG. This choice determines how OpenRAG authenticates with OpenSearch and controls access to

documents.

> **INFO**
>
> You must use **Advanced Setup** if you want to use OAuth connectors to upload documents from cloud storage.

If OpenRAG detects OAuth credentials during setup, it recommends **Advanced Setup** in the TUI.

Basic setup

1. In the TUI, click **Basic Setup** or press `1`.
2. Enter administrator passwords for the OpenRAG OpenSearch and Langflow services, or click **Generate Passwords** to generate passwords automatically.

   The OpenSearch password is required.

   The Langflow password is recommended but optional. If the Langflow password is empty, the Langflow server starts without authentication enabled. For more information, see Langflow settings.
3. Optional: Enter your OpenAI API key, or leave this field empty if you want to configure model provider credentials later during application onboarding.
4. Click **Save Configuration**.

   Your passwords and API key, if provided, are stored in the `.env` file in your OpenRAG installation directory. If you modified any credentials that were pulled from an existing `.env` file, those values are updated in the `.env` file.
5. Click **Start All Services** to start the OpenRAG services that run in containers.

   This process can take some time while OpenRAG pulls and runs the container images. If all services start successfully, the TUI prints a confirmation message:

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

6. Under **Native Services**, click **Start** to start the Docling service.

7. Launch the OpenRAG application:

    - From the TUI main menu, click **Open App**.
    - In your browser, navigate to `localhost:3000`.
8. Continue with application onboarding.

Advanced setup

1. In the TUI, click **Advanced Setup** or press `2`.

2. Enter administrator passwords for the OpenRAG OpenSearch and Langflow services, or click **Generate Passwords** to generate passwords automatically.

    The OpenSearch password is required.

    The Langflow password is recommended but optional. If the Langflow password is empty, the Langflow server starts without authentication enabled. For more information, see Langflow settings.

3. Optional: Enter your OpenAI API key, or leave this field empty if you want to configure model provider credentials later during application onboarding.

4. 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**.

   Your passwords, API key, and OAuth credentials, if provided, are stored in the `.env` file in your OpenRAG installation directory. If you modified any credentials that were pulled from an existing `.env` file, those values are updated in the `.env` file.

7. Click **Start All Services** to start the OpenRAG services that run in containers.

   This process can take some time while OpenRAG pulls and runs the container images. If all services start successfully, the TUI prints a confirmation message:

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

8. Under **Native Services**, click **Start** to start the Docling service.

9. Launch the OpenRAG application:

   - From the TUI main menu, click **Open App**.
   - 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. If required, you can edit the following additional environment variables. Only change these variables if your OpenRAG deployment has a non-default network configuration, such as 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 for the following OpenRAG OAuth connector endpoints:

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

12. Continue with application onboarding.

Application onboarding

The first time you start the OpenRAG application, you must complete application onboarding to select language and embedding models that are essential for OpenRAG features like the **Chat**.

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 embedding 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 the embedding model.

1. Enter your Anthropic API key, or enable **Get API key from environment variable** to pull the key from your OpenRAG `.env` file.

   If you haven't set `ANTHROPIC_API_KEY` in your `.env` file, you must enter the key manually.

2. Under **Advanced settings**, select the language model that you want to use.

3. Click **Complete**.

4. Select a provider for embeddings, provide the required information, and then select the embedding model you want to use. For information about another provider's credentials and settings, see the instructions for that provider.

5. Click **Complete**.

   After you configure the embedding model, OpenRAG uses your credentials and models to ingest some initial documents. This tests the connection, and it allows you to ask OpenRAG about itself in the **Chat**. If there is a problem with the model configuration, an error occurs and you are redirected back to application onboarding. Verify that the credential is valid and has access to the selected model, and then click **Complete** to retry ingestion.

6. Continue through the overview slides for a brief introduction to OpenRAG, or click → **Skip overview**. The overview demonstrates some basic functionality that is covered in the quickstart and in other parts of the OpenRAG documentation.

IBM watsonx.ai

1. Use the values from your IBM watsonx deployment for the **watsonx.ai API Endpoint**, **IBM Project ID**, and **IBM API key** fields.

2. Under **Advanced settings**, select the language model that you want to use.

3. Click **Complete**.

4. Select a provider for embeddings, provide the required information, and then select the embedding model you want to use. For information about another provider's credentials and settings, see the instructions for that provider.

5. Click **Complete**.

   After you configure the embedding model, OpenRAG uses your credentials and models to ingest some initial documents. This tests the connection, and it allows you to ask OpenRAG about itself in the **Chat**. If there is a problem with the model configuration, an error occurs and you are redirected back to application onboarding. Verify that the credentials are valid and have access to the selected model, and then click **Complete** to retry ingestion.

6. Continue through the overview slides for a brief introduction to OpenRAG, or click → **Skip overview**. The overview demonstrates some basic functionality that is covered in the quickstart and in other parts of the OpenRAG documentation.

Ollama

> **INFO**
> Ollama isn't installed with OpenRAG. You must install it separately if you want to use Ollama as a model provider.

Using Ollama as your language and embedding model provider offers greater flexibility and configuration options for hosting models, but it can be advanced for new users. The recommendations given here are a reasonable starting point for users with at least one GPU and experience running LLMs locally.

The OpenRAG team recommends the OpenAI `gpt-oss:20b` lanuage model and the `nomic-embed-text` embedding model. However, `gpt-oss:20b` uses 16GB of RAM, so consider using Ollama Cloud or running Ollama on a remote machine.

1. Install Ollama locally or on a remote server or run models in Ollama Cloud.

   If you are running a remote server, it must be accessible from your OpenRAG deployment.

2. In OpenRAG onboarding, connect to your Ollama server:

   - **Local Ollama server**: Enter your Ollama server's base URL and port. The default Ollama server address is `http://localhost:11434`.
   - **Ollama Cloud**: Because Ollama Cloud models run at the same address as a local Ollama server and automatically offload to Ollama's cloud service, you can use the same base URL and port as you would for a local Ollama server. The default address is `http://localhost:11434`.
   - **Remote server**: Enter your remote Ollama server's base URL and port, such as `http://your-remote-server:11434`.

   If the connection succeeds, OpenRAG populates the model lists with the server's available models.

3. Select the model that your Ollama server is running.

   Language model and embedding model selections are independent. You can use the same or different servers for each model.

   To use different providers for each model, you must configure both providers, and select the relevant model for each provider.

4. Click **Complete**.

   After you configure the embedding model, OpenRAG uses the address and models to ingest some initial documents. This tests the connection, and it allows you to ask OpenRAG about itself in the **Chat**. If there is a problem with the model configuration, an error occurs and you are redirected back to application onboarding. Verify that the server address is valid, and that the selected model is running on the server. Then, click **Complete** to retry ingestion.

5. Continue through the overview slides for a brief introduction to OpenRAG, or click **Skip overview**. The overview demonstrates some basic functionality that is covered in the quickstart and in other parts of the OpenRAG documentation.

OpenAI (default)

1. Enter your OpenAI API key, or enable **Get API key from environment variable** to pull the key from your OpenRAG `.env` file.

   If you entered an OpenAI API key during setup, enable **Get API key from environment variable**.

2. Under **Advanced settings**, select the language model that you want to use.

3. Click **Complete**.

4. Select a provider for embeddings, provide the required information, and then select the embedding model you want to use. For information about another provider's credentials and settings, see the instructions for that provider.

5. Click **Complete**.

After you configure the embedding model, OpenRAG uses your credentials and models to ingest some initial documents. This tests the connection, and it allows you to ask OpenRAG about itself in the **Chat**. If there is a problem with the model configuration, an error occurs and you are redirected back to application onboarding. Verify that the credential is valid and has access to the selected model, and then click **Complete** to retry ingestion.

6. Continue through the overview slides for a brief introduction to OpenRAG, or click → **Skip overview**. The overview demonstrates some basic functionality that is covered in the quickstart and in other parts of the OpenRAG documentation.

Next steps

- Try some of OpenRAG's core features in the quickstart.
- Learn how to manage OpenRAG services.
- Upload documents, and then use the **Chat** to explore your data.

Invoke OpenRAG with uvx

For guided configuration and simplified service management, install OpenRAG with services managed by the Terminal User Interface (TUI).

You can use `uvx` to invoke OpenRAG outside of a Python project or without modifying your project's dependencies.

> **TIP**
>
> The automatic installer script also uses `uvx` to install OpenRAG.

This installation method is best for testing OpenRAG by running it outside of a Python project. For other installation methods, see Select an installation method.

Prerequisites

- For Microsoft Windows, you must use the Windows Subsystem for Linux (WSL). See Install OpenRAG on Windows before proceeding.
- Gather the credentials and connection details for your preferred model providers. You must have access to at least one language model and one embedding model. If a provider offers both types, you can use the same provider for both models. If a provider offers only one type, you must select two providers.
    - **OpenAI**: Create an OpenAI API key.
    - **Anthropic**: Create an Anthropic API key. Anthropic provides language models only; you must select an additional provider for embeddings.
    - **IBM watsonx.ai**: Get your watsonx.ai API endpoint, IBM project ID, and IBM API key from your watsonx deployment.
    - **Ollama**: Deploy an Ollama instance and models locally, in the cloud, or on a remote server, and then get your Ollama server's base URL and the names of the models that you want to use.
- 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 Python version 3.13 or later.

- Install uv.
- Install Podman (recommended) or Docker.
- Install `podman-compose` or Docker Compose. To use Docker Compose with Podman, you must alias Docker Compose commands to Podman commands.

Install and run OpenRAG with uvx

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

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

2. Optional: If you want to use a pre-populated `.env` file for OpenRAG, copy it to this directory before invoking OpenRAG.

3. Invoke OpenRAG:

   ```
   uvx openrag
   ```

   You can invoke a specific version using any of the `uvx` version specifiers, such as `--from`:

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

   Invoking OpenRAG with `uvx openrag` creates a cached, ephemeral environment for the TUI in your local `uv` cache. By invoking OpenRAG in a specific directory, your OpenRAG configuration files and data are stored separately from the `uv` cache. Clearing the `uv` cache doesn't remove your entire OpenRAG installation. After clearing the cache, you can re-invoke OpenRAG (`uvx openrag`) to restart the TUI with your preserved configuration and data.

If you encounter errors during installation, see Troubleshoot OpenRAG.

Set up OpenRAG with the TUI

When you install OpenRAG with `uvx`, you manage the OpenRAG services with the Terminal User Interface (TUI). The TUI guides you through the initial configuration process before you start the OpenRAG services.

Your OpenRAG configuration is stored in a `.env` file that is created automatically in the OpenRAG installation directory. If OpenRAG detects an existing `.env` file, the TUI automatically populates those values during setup and onboarding.

Container definitions are stored in the `docker-compose` files in the OpenRAG installation directory.

With `uvx`, the OpenRAG `.env` and `docker-compose` files are stored in the directory where you invoked OpenRAG.

You can use either **Basic Setup** or **Advanced Setup** to configure OpenRAG. This choice determines how OpenRAG authenticates with OpenSearch and controls access to documents.

> **INFO**
>
> You must use **Advanced Setup** if you want to use OAuth connectors to upload documents from cloud storage.

If OpenRAG detects OAuth credentials during setup, it recommends **Advanced Setup** in the TUI.

Basic setup

1. In the TUI, click **Basic Setup** or press `1`.
2. Enter administrator passwords for the OpenRAG OpenSearch and Langflow services, or click **Generate Passwords** to generate passwords automatically.

   The OpenSearch password is required.

   The Langflow password is recommended but optional. If the Langflow password is empty, the Langflow server starts without authentication enabled. For more

information, see Langflow settings.

3. Optional: Enter your OpenAI API key, or leave this field empty if you want to configure model provider credentials later during application onboarding.

4. Click **Save Configuration**.

   Your passwords and API key, if provided, are stored in the `.env` file in your OpenRAG installation directory. If you modified any credentials that were pulled from an existing `.env` file, those values are updated in the `.env` file.

5. Click **Start All Services** to start the OpenRAG services that run in containers.

   This process can take some time while OpenRAG pulls and runs the container images. If all services start successfully, the TUI prints a confirmation message:

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

6. Under **Native Services**, click **Start** to start the Docling service.

7. Launch the OpenRAG application:

   - From the TUI main menu, click **Open App**.
   - In your browser, navigate to `localhost:3000`.

8. Continue with application onboarding.

Advanced setup

1. In the TUI, click **Advanced Setup** or press `2`.

2. Enter administrator passwords for the OpenRAG OpenSearch and Langflow services, or click **Generate Passwords** to generate passwords automatically.

   The OpenSearch password is required.

   The Langflow password is recommended but optional. If the Langflow password is empty, the Langflow server starts without authentication enabled. For more information, see Langflow settings.

3. Optional: Enter your OpenAI API key, or leave this field empty if you want to configure model provider credentials later during application onboarding.

4. 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**.

    Your passwords, API key, and OAuth credentials, if provided, are stored in the `.env` file in your OpenRAG installation directory. If you modified any credentials that were pulled from an existing `.env` file, those values are updated in the `.env` file.

7. Click **Start All Services** to start the OpenRAG services that run in containers.

    This process can take some time while OpenRAG pulls and runs the container images. If all services start successfully, the TUI prints a confirmation message:

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

8. Under **Native Services**, click **Start** to start the Docling service.

9. Launch the OpenRAG application:

    - From the TUI main menu, click **Open App**.
    - 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. If required, you can edit the following additional environment variables. Only change these variables if your OpenRAG deployment has a non-default network configuration, such as 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 for the following OpenRAG OAuth connector endpoints:

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

12. Continue with application onboarding.

Application onboarding

The first time you start the OpenRAG application, you must complete application onboarding to select language and embedding models that are essential for OpenRAG features like the **Chat**.

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 embedding 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 the embedding model.

1. Enter your Anthropic API key, or enable **Get API key from environment variable** to pull the key from your OpenRAG `.env` file.

   If you haven't set `ANTHROPIC_API_KEY` in your `.env` file, you must enter the key manually.

2. Under **Advanced settings**, select the language model that you want to use.

3. Click **Complete**.

4. Select a provider for embeddings, provide the required information, and then select the embedding model you want to use. For information about another provider's credentials and settings, see the instructions for that provider.

5. Click **Complete**.

   After you configure the embedding model, OpenRAG uses your credentials and models to ingest some initial documents. This tests the connection, and it allows you to ask OpenRAG about itself in the **Chat**. If there is a problem with the model configuration, an error occurs and you are redirected back to application onboarding. Verify that the credential is valid and has access to the selected model, and then click **Complete** to retry ingestion.

6. Continue through the overview slides for a brief introduction to OpenRAG, or click **Skip overview**. The overview demonstrates some basic functionality that is covered in the quickstart and in other parts of the OpenRAG documentation.

IBM watsonx.ai

1. Use the values from your IBM watsonx deployment for the **watsonx.ai API Endpoint**, **IBM Project ID**, and **IBM API key** fields.

2. Under **Advanced settings**, select the language model that you want to use.

3. Click **Complete**.

4. Select a provider for embeddings, provide the required information, and then select the embedding model you want to use. For information about another provider's credentials and settings, see the instructions for that provider.

5. Click **Complete**.

   After you configure the embedding model, OpenRAG uses your credentials and models to ingest some initial documents. This tests the connection, and it allows you to ask OpenRAG about itself in the **Chat**. If there is a problem with the model configuration, an error occurs and you are redirected back to application onboarding. Verify that the credentials are valid and have access to the selected model, and then click **Complete** to retry ingestion.

6. Continue through the overview slides for a brief introduction to OpenRAG, or click **Skip overview** →. The overview demonstrates some basic functionality that is covered in the quickstart and in other parts of the OpenRAG documentation.

Ollama

> **INFO**
>
> Ollama isn't installed with OpenRAG. You must install it separately if you want to use Ollama as a model provider.

Using Ollama as your language and embedding model provider offers greater flexibility and configuration options for hosting models, but it can be advanced for new users. The recommendations given here are a reasonable starting point for users with at least one GPU and experience running LLMs locally.

The OpenRAG team recommends the OpenAI `gpt-oss:20b` lanuage model and the `nomic-embed-text` embedding model. However, `gpt-oss:20b` uses 16GB of RAM, so consider using Ollama Cloud or running Ollama on a remote machine.

1. Install Ollama locally or on a remote server or run models in Ollama Cloud.

   If you are running a remote server, it must be accessible from your OpenRAG deployment.

2. In OpenRAG onboarding, connect to your Ollama server:

   - **Local Ollama server**: Enter your Ollama server's base URL and port. The default Ollama server address is `http://localhost:11434`.
   - **Ollama Cloud**: Because Ollama Cloud models run at the same address as a local Ollama server and automatically offload to Ollama's cloud service, you can use the same base URL and port as you would for a local Ollama server. The default address is `http://localhost:11434`.
   - **Remote server**: Enter your remote Ollama server's base URL and port, such as `http://your-remote-server:11434`.

   If the connection succeeds, OpenRAG populates the model lists with the server's available models.

3. Select the model that your Ollama server is running.

   Language model and embedding model selections are independent. You can use the same or different servers for each model.

   To use different providers for each model, you must configure both providers, and select the relevant model for each provider.

4. Click **Complete**.

   After you configure the embedding model, OpenRAG uses the address and models to ingest some initial documents. This tests the connection, and it allows you to ask OpenRAG about itself in the **Chat**. If there is a problem with the model configuration, an error occurs and you are redirected back to application onboarding. Verify that the server address is valid, and that the selected model is running on the server. Then, click **Complete** to retry ingestion.

5. Continue through the overview slides for a brief introduction to OpenRAG, or click → **Skip overview**. The overview demonstrates some basic functionality that is covered in the quickstart and in other parts of the OpenRAG documentation.

OpenAI (default)

1. Enter your OpenAI API key, or enable **Get API key from environment variable** to pull the key from your OpenRAG `.env` file.

   If you entered an OpenAI API key during setup, enable **Get API key from environment variable**.

2. Under **Advanced settings**, select the language model that you want to use.

3. Click **Complete**.

4. Select a provider for embeddings, provide the required information, and then select the embedding model you want to use. For information about another provider's credentials and settings, see the instructions for that provider.

5. Click **Complete**.

   After you configure the embedding model, OpenRAG uses your credentials and models to ingest some initial documents. This tests the connection, and it allows you to ask OpenRAG about itself in the **Chat**. If there is a problem with the model configuration, an error occurs and you are redirected back to application onboarding. Verify that the credential is valid and has access to the selected model, and then click **Complete** to retry ingestion.

6. Continue through the overview slides for a brief introduction to OpenRAG, or click → **Skip overview**. The overview demonstrates some basic functionality that is covered in the quickstart and in other parts of the OpenRAG documentation.

Next steps

- Try some of OpenRAG's core features in the quickstart.
- Learn how to manage OpenRAG services.
- Upload documents, and then use the **Chat** to explore your data.

Install OpenRAG on Microsoft Windows

If you're using Windows, you must install OpenRAG within the Windows Subsystem for Linux (WSL).

Nested virtualization isn't supported

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.

Install OpenRAG in the WSL

1. Install WSL with an 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.

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

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

4. 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.

   The Docker Desktop WSL integration makes Docker available within your WSL distribution. You don't need to install Docker or Podman separately in your WSL distribution before you install OpenRAG.

5. Install and run OpenRAG from within your WSL Ubuntu distribution. You can install OpenRAG in your WSL distribution using any of the OpenRAG installation methods.

Troubleshoot OpenRAG in WSL

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.

Deploy OpenRAG with self-managed services

To manage your own OpenRAG services, deploy OpenRAG with Docker or Podman.

Use this installation method if you don't want to use the Terminal User Interface (TUI), or you need to run OpenRAG in an environment where using the TUI is unfeasible.

Prerequisites

- For Microsoft Windows, you must use the Windows Subsystem for Linux (WSL). See Install OpenRAG on Windows before proceeding.
- Gather the credentials and connection details for your preferred model providers. You must have access to at least one language model and one embedding model. If a provider offers both types, you can use the same provider for both models. If a provider offers only one type, you must select two providers.
    - **OpenAI**: Create an OpenAI API key.
    - **Anthropic**: Create an Anthropic API key. Anthropic provides language models only; you must select an additional provider for embeddings.
    - **IBM watsonx.ai**: Get your watsonx.ai API endpoint, IBM project ID, and IBM API key from your watsonx deployment.
    - **Ollama**: Deploy an Ollama instance and models locally, in the cloud, or on a remote server, and then get your Ollama server's base URL and the names of the models that you want to use.
- 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 Python version 3.13 or later.
- Install uv.
- Install Podman (recommended) or Docker.
- Install `podman-compose` or Docker Compose. To use Docker Compose with Podman, you must alias Docker Compose commands to Podman commands.

Prepare your deployment

1. Clone the OpenRAG repository:

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

2. Change to the root of the cloned repository:

```
cd openrag
```

3. Install dependencies:

```
uv sync
```

4. Create a `.env` file at the root of the cloned repository.

   You can create an empty file or copy the repository's `.env.example` file. The example file contains some of the OpenRAG environment variables to get you started with configuring your deployment.

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

5. Edit the `.env` file to configure your deployment using OpenRAG environment variables. The OpenRAG Docker Compose files pull values from your `.env` file to configure the OpenRAG containers. The following variables are required or recommended:

   - **`OPENSEARCH_PASSWORD` (Required)**: Sets the OpenSearch administrator password. It must adhere to the OpenSearch password complexity requirements.

   - **`LANGFLOW_SUPERUSER`**: The username for the Langflow administrator user. Defaults to `admin` if not set.

- `LANGFLOW_SUPERUSER_PASSWORD` **(Strongly recommended)**: Sets the Langflow administrator password, and determines the Langflow server's default authentication mode. If not set, the Langflow server starts without authentication enabled. For more information, see Langflow settings.

- `LANGFLOW_SECRET_KEY` **(Strongly recommended)**: A secret encryption key for internal Langflow operations. It is recommended to generate your own Langflow secret key. If not set, Langflow generates a secret key automatically.

- **Model provider credentials**: Provide credentials for your preferred model providers. If not set in the `.env` file, you must configure at least one provider during application onboarding.

  - `OPENAI_API_KEY`
  - `ANTHROPIC_API_KEY`
  - `OLLAMA_ENDPOINT`
  - `WATSONX_API_KEY`
  - `WATSONX_ENDPOINT`
  - `WATSONX_PROJECT_ID`
- **OAuth provider credentials**: To upload documents from external storage, such as Google Drive, set the required OAuth credentials for the connectors that you want to use. You can manage OAuth credentials later, but it is recommended to configure them during initial set up so you don't have to rebuild the containers.

  - **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.

For more information and variables, see OpenRAG environment variables.

Start services

1. Start `docling serve` on port 5001 on the host machine:

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

   Docling cannot run inside a Docker container due to system-level dependencies, so you must manage it as a separate service on the host machine. For more information, see Stop, start, and inspect native services.

   This port is required to deploy OpenRAG successfully; don't use a different port. Additionally, this enables the MLX framework for accelerated performance on Apple Silicon Mac machines.

2. Confirm `docling serve` is running.

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

   If `docling serve` is running, the output includes the status, address, and process ID (PID):

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

3. Deploy the OpenRAG containers locally using the appropriate Docker Compose file for your environment. Both files deploy the same services.

   - `docker-compose.yml`: If your host machine has an NVIDIA GPU with CUDA support and compatible NVIDIA drivers, you can use this file to deploy OpenRAG with accelerated processing.

     Docker

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

Podman

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

- `docker-compose-cpu.yml`: If your host machine doesn't have NVIDIA GPU support, use this file for a CPU-only OpenRAG deployment.

Docker

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

Podman

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

4. Wait for the OpenRAG containers to start, and then confirm that all containers are running:

Docker

```
docker compose ps
```

Podman

```
podman compose ps
```

The OpenRAG Docker Compose files deploy the following 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 user interaction.

Container Name	Default address	Purpose
Langflow	http://localhost:7860	AI workflow engine.
OpenSearch	http://localhost:9200	Datastore for knowledge.
OpenSearch Dashboards	http://localhost:5601	OpenSearch database administration interface.

When the containers are running, you can access your OpenRAG services at their addresses.

5. Access the OpenRAG frontend at `http://localhost:3000`, and then continue with application onboarding.

Application onboarding

The first time you start the OpenRAG application, you must complete application onboarding to select language and embedding models that are essential for OpenRAG features like the **Chat**.

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 embedding 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 the embedding model.

1. Enter your Anthropic API key, or enable **Get API key from environment variable** to pull the key from your OpenRAG `.env` file.

If you haven't set `ANTHROPIC_API_KEY` in your `.env` file, you must enter the key manually.

2. Under **Advanced settings**, select the language model that you want to use.

3. Click **Complete**.

4. Select a provider for embeddings, provide the required information, and then select the embedding model you want to use. For information about another provider's credentials and settings, see the instructions for that provider.

5. Click **Complete**.

   After you configure the embedding model, OpenRAG uses your credentials and models to ingest some initial documents. This tests the connection, and it allows you to ask OpenRAG about itself in the **Chat**. If there is a problem with the model configuration, an error occurs and you are redirected back to application onboarding. Verify that the credential is valid and has access to the selected model, and then click **Complete** to retry ingestion.

6. Continue through the overview slides for a brief introduction to OpenRAG, or click **Skip overview**. The overview demonstrates some basic functionality that is covered in the quickstart and in other parts of the OpenRAG documentation.

IBM watsonx.ai

1. Use the values from your IBM watsonx deployment for the **watsonx.ai API Endpoint**, **IBM Project ID**, and **IBM API key** fields.

2. Under **Advanced settings**, select the language model that you want to use.

3. Click **Complete**.

4. Select a provider for embeddings, provide the required information, and then select the embedding model you want to use. For information about another provider's credentials and settings, see the instructions for that provider.

5. Click **Complete**.

After you configure the embedding model, OpenRAG uses your credentials and models to ingest some initial documents. This tests the connection, and it allows you to ask OpenRAG about itself in the **Chat**. If there is a problem with the model configuration, an error occurs and you are redirected back to application onboarding. Verify that the credentials are valid and have access to the selected model, and then click **Complete** to retry ingestion.

6. Continue through the overview slides for a brief introduction to OpenRAG, or click → **Skip overview**. The overview demonstrates some basic functionality that is covered in the quickstart and in other parts of the OpenRAG documentation.

Ollama

> **INFO**
> Ollama isn't installed with OpenRAG. You must install it separately if you want to use Ollama as a model provider.

Using Ollama as your language and embedding model provider offers greater flexibility and configuration options for hosting models, but it can be advanced for new users. The recommendations given here are a reasonable starting point for users with at least one GPU and experience running LLMs locally.

The OpenRAG team recommends the OpenAI `gpt-oss:20b` lanuage model and the `nomic-embed-text` embedding model. However, `gpt-oss:20b` uses 16GB of RAM, so consider using Ollama Cloud or running Ollama on a remote machine.

1. Install Ollama locally or on a remote server or run models in Ollama Cloud.

   If you are running a remote server, it must be accessible from your OpenRAG deployment.

2. In OpenRAG onboarding, connect to your Ollama server:

   - **Local Ollama server**: Enter your Ollama server's base URL and port. The default Ollama server address is `http://localhost:11434`.
   - **Ollama Cloud**: Because Ollama Cloud models run at the same address as a local Ollama server and automatically offload to Ollama's cloud service, you can

use the same base URL and port as you would for a local Ollama server. The default address is `http://localhost:11434`.

- **Remote server**: Enter your remote Ollama server's base URL and port, such as `http://your-remote-server:11434`.

If the connection succeeds, OpenRAG populates the model lists with the server's available models.

3. Select the model that your Ollama server is running.

   Language model and embedding model selections are independent. You can use the same or different servers for each model.

   To use different providers for each model, you must configure both providers, and select the relevant model for each provider.

4. Click **Complete**.

   After you configure the embedding model, OpenRAG uses the address and models to ingest some initial documents. This tests the connection, and it allows you to ask OpenRAG about itself in the **Chat**. If there is a problem with the model configuration, an error occurs and you are redirected back to application onboarding. Verify that the server address is valid, and that the selected model is running on the server. Then, click **Complete** to retry ingestion.

5. Continue through the overview slides for a brief introduction to OpenRAG, or click **Skip overview**. The overview demonstrates some basic functionality that is covered in the quickstart and in other parts of the OpenRAG documentation.

OpenAI (default)

1. Enter your OpenAI API key, or enable **Get API key from environment variable** to pull the key from your OpenRAG `.env` file.

   If you entered an OpenAI API key during setup, enable **Get API key from environment variable**.

2. Under **Advanced settings**, select the language model that you want to use.

3. Click **Complete**.

4. Select a provider for embeddings, provide the required information, and then select the embedding model you want to use. For information about another provider's credentials and settings, see the instructions for that provider.

5. Click **Complete**.

   After you configure the embedding model, OpenRAG uses your credentials and models to ingest some initial documents. This tests the connection, and it allows you to ask OpenRAG about itself in the **Chat**. If there is a problem with the model configuration, an error occurs and you are redirected back to application onboarding. Verify that the credential is valid and has access to the selected model, and then click **Complete** to retry ingestion.

6. Continue through the overview slides for a brief introduction to OpenRAG, or click **Skip overview** → . The overview demonstrates some basic functionality that is covered in the quickstart and in other parts of the OpenRAG documentation.

Next steps

- Try some of OpenRAG's core features in the quickstart.
- Learn how to manage OpenRAG services.
- Upload documents, and then use the **Chat** to explore your data.

Upgrade OpenRAG

Use these steps to upgrade your OpenRAG deployment to the latest version or a specific version.

Export customized flows before upgrading

If you modified the built-in flows or created custom flows in your OpenRAG Langflow instance, export your flows before upgrading. This ensure that you won't lose your flows after upgrading, and you can reference the exported flows if there are any breaking changes in the new version.

Upgrade TUI-managed installations

To upgrade OpenRAG, you need to upgrade the OpenRAG Python package, and then upgrade the OpenRAG containers.

Upgrading the Python package also upgrades Docling by bumping the dependency in `pyproject.toml`.

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

1. To check for updates, open the TUI's **Status** menu (`3`), and then click **Upgrade**.
2. If there is an update, stop all OpenRAG services. In the **Status** menu, click **Stop Services**.
3. Upgrade the OpenRAG Python package to the latest version from PyPI. The commands to upgrade the package depend on how you installed OpenRAG.
    - Use these steps to upgrade the Python package if you installed OpenRAG using the installer script or `uvx`:

        a. Navigate to your OpenRAG workspace directory:

        ```
        cd openrag-workspace
        ```

b. Upgrade the OpenRAG package:

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

You can invoke a specific version using any of the `uvx` version specifiers, such as `--from`:

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

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

a. Navigate to your project directory:

```
cd YOUR_PROJECT_NAME
```

b. Update OpenRAG to the latest version:

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

To upgrade to a specific version:

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

c. Start the OpenRAG TUI:

```
uv run openrag
```

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

a. Activate your virtual environment.

b. Upgrade OpenRAG:

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

To upgrade to a specific version:

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

c. Start the OpenRAG TUI:

```
uv run openrag
```

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

   When you start services 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`.

   > **PIN CONTAINER VERSIONS**
   >
   > 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. Under **Native Services**, click **Start** to start the Docling service.

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

Upgrade self-managed containers

To fetch and apply the latest container images while preserving your OpenRAG data, run the commands for your container management tool. By default, OpenRAG's `docker-compose` files pull the latest container images.

Docker

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

Podman

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

See also

- Manage OpenRAG services
- Troubleshoot OpenRAG

Reinstall OpenRAG

You can reset your OpenRAG deployment to its initial state by recreating the containers and deleting accessory data like the `.env` file and ingested documents.

> **WARNING**
> These are destructive operations that reset your OpenRAG deployment to an initial state. Destroyed containers and deleted data are lost and cannot be recovered after running these operations.

Export customized flows before reinstalling

If you modified the built-in flows or created custom flows in your OpenRAG Langflow instance, and you want to preserve those changes, export your flows before reinstalling OpenRAG.

Reinstall TUI-managed containers

1. In the TUI's **Status** menu (`3`), click **Factory Reset** to destroy your OpenRAG containers and some related data.

   > **WARNING**
   > This is a destructive action that does the following:
   >
   > - Destroys all OpenRAG containers, volumes, and local images with `docker compose down --volumes --remove-orphans --rmi local`.
   > - Prunes any additional Docker objects with `docker system prune -f`.
   > - Deletes the contents of OpenRAG's `config` and `./opensearch-data` directories.
   > - Deletes the `conversations.json` file.
   >
   > Destroyed containers and deleted data are lost and cannot be recovered after running this operation.
   >
   > This operation *doesn't* remove the `.env` file or the contents of the `./openrag-documents` directory.

2. Exit the TUI with `q`.

3. Optional: Remove data that wasn't deleted by the **Factory Reset** operation. For a completely fresh installation, delete all of this data.

    - **OpenRAG's `.env` file**: Contains your OpenRAG configuration, including OpenRAG passwords, API keys, OAuth settings, and other environment variables. If you delete this file, OpenRAG automatically generates a new one after you repeat the setup and onboarding process. Alternatively, you can add a prepopulated `.env` file to your OpenRAG installation directory before restarting OpenRAG.
    - **The contents of the `./openrag-documents` directory**: Contains documents that you uploaded to OpenRAG. Delete these files to prevent documents from being reingested to your knowledge base after restarting OpenRAG. However, you might want to preserve OpenRAG's default documents.
4. Restart the TUI with `uv run openrag` or `uvx openrag`.

5. Repeat the setup process to configure OpenRAG and restart all services. Then, launch the OpenRAG app and repeat application onboarding.

    If OpenRAG detects a `.env` file during setup and onboarding, it automatically populates any OpenRAG passwords, OAuth credentials, and onboarding configuration set in that file.

Reinstall with Docker Compose or Podman Compose

1. Destroy the containers, volumes, and local images, and then remove (prune) any additional Podman objects:

    Docker

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

    Podman

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

2. Optional: Remove data that wasn't deleted by the previous commands:
    - OpenRAG's `.env` file
    - The contents of OpenRAG's `config` directory
    - The contents of the `./openrag-documents` directory
    - The contents of the `./opensearch-data` directory
    - The `conversations.json` file
3. If you deleted the `.env` file, prepare a new `.env` before redeploying the containers. For more information, see Deploy OpenRAG with self-managed services.
4. Redeploy OpenRAG:

    Docker

    ```
    docker compose up -d
    ```

    Podman

    ```
    podman compose up -d
    ```

5. Launch the OpenRAG app, and then repeat application onboarding.

Step-by-step reinstallation with Docker or Podman

Use these commands for step-by-step container removal and cleanup:

1. Stop all running containers:

    Docker

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

Podman

```
podman stop --all
```

2. Remove all containers, including stopped containers:

Docker

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

Podman

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

3. Remove all images:

Docker

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

Podman

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

4. Remove all volumes:

Docker

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

Podman

```
```

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

5. Remove all networks except the default network:

Docker

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

Podman

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

6. Clean up any leftover data:

Docker

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

Podman

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

7. Optional: Remove data that wasn't deleted by the previous commands:

    - OpenRAG's `.env` file
    - The contents of OpenRAG's `config` directory
    - The contents of the `./openrag-documents` directory
    - The contents of the `./opensearch-data` directory
    - The `conversations.json` file

8. Redeploy OpenRAG.

Remove OpenRAG

> **TIP**
>
> If you want to reset your OpenRAG containers without removing OpenRAG entirely, see Reset OpenRAG containers and Reinstall OpenRAG.

Uninstall TUI-managed deployments

If you used the automated installer script or `uvx` to install OpenRAG, clear your `uv` cache (`uv cache clean`) to remove the TUI environment, and then delete the directory containing your OpenRAG configuration files and data (where you would invoke OpenRAG).

If you used `uv` to install OpenRAG, run `uv remove openrag` in your Python project.

Uninstall self-managed deployments

For self-managed services, destroy the containers, prune any additional Docker objects, shut down the Docling service, and delete any remaining OpenRAG files.

Uninstall with Docker Compose or Podman Compose

1. Destroy the containers, volumes, and local images, and then remove (prune) any additional Docker objects:

   Docker

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

   Podman

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

2. Remove data that wasn't deleted by the previous commands:

- OpenRAG's `.env` file
- The contents of OpenRAG's `config` directory
- The contents of the `./openrag-documents` directory
- The contents of the `./opensearch-data` directory
- The `conversations.json` file

3. Stop `docling-serve`:

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

Step-by-step removal and cleanup with Docker or Podman

Use these commands for step-by-step container removal and cleanup:

1. Stop all running containers:

Docker

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

Podman

```
podman stop --all
```

2. Remove all containers, including stopped containers:

Docker

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

Podman

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

3. Remove all images:

Docker

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

Podman

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

4. Remove all volumes:

Docker

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

Podman

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

5. Remove all networks except the default network:

Docker

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

Podman

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

6. Clean up any leftover data:

Docker

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

Podman

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

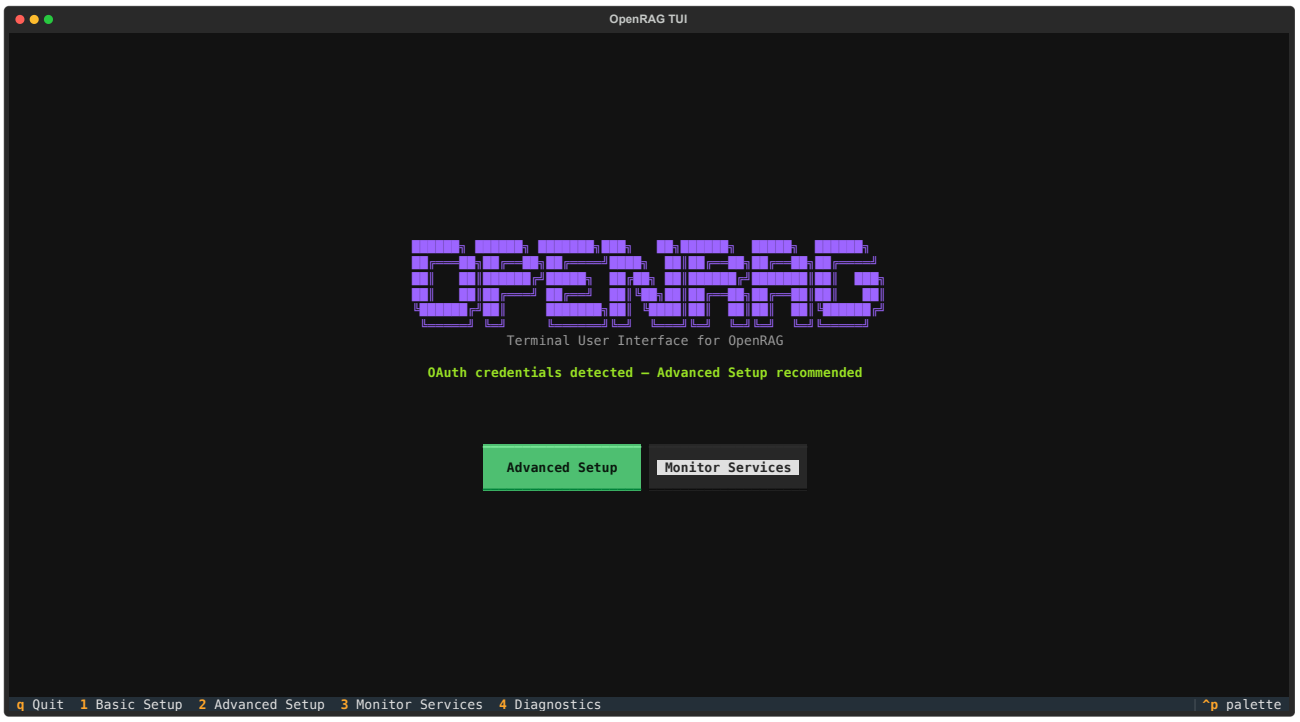
7. Remove data that wasn't deleted by the previous commands:

    - OpenRAG's `.env` file
    - The contents of OpenRAG's `config` directory
    - The contents of the `./openrag-documents` directory
    - The contents of the `./opensearch-data` directory
    - The `conversations.json` file
8. Stop `docling-serve`:

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

Use the TUI

The OpenRAG Terminal User Interface (TUI) provides a simplified and guided experience for configuring, managing, and monitoring your OpenRAG deployment directly from the terminal.



If you install OpenRAG with the automatic installer script, `uv`, or `uvx`, you use the TUI to manage your OpenRAG deployment. The TUI guides you through the initial setup, automatically manages your OpenRAG `.env` and `docker-compose` files, and provides convenient access to service management controls.

In contrast, when you deploy OpenRAG with self-managed services, you must manually configure OpenRAG by preparing a `.env` file and using Docker or Podman commands to deploy and manage your OpenRAG services.

Access the TUI

If you installed OpenRAG with `uv`, access the TUI with `uv run openrag`.

If you installed OpenRAG with the automatic installer script or `uvx`, access the TUI with `uvx openrag`.

Manage services with the TUI

Use the TUI's **Status** menu (`3`) and **Diagnostics** menu (`4`) to access controls and information for your OpenRAG services. For more information, see Manage OpenRAG services.

Exit the OpenRAG TUI

To exit the OpenRAG TUI, go to the TUI main menu, and then press `q`.

Your OpenRAG containers continue to run until they are stopped.

To restart the TUI, see Access the TUI.

Manage OpenRAG containers and services

Service management is an essential part of maintaining your OpenRAG deployment.

Most OpenRAG services run in containers. However, some services, like Docling, run directly on the local machine.

If you installed OpenRAG with the automated installer script, `uv`, or `uvx`, you can use the Terminal User Interface (TUI) to manage your OpenRAG configuration and services.

For self-managed deployments, run Docker or Podman commands to manage your OpenRAG services.

Monitor services

- **TUI Status menu**: In the **Status** menu (`3`), you can access streaming logs for all OpenRAG services. Select the service you want to view, and then press `l`. To copy the logs, click **Copy to Clipboard**.
- **TUI Diagnostics menu**: The TUI's **Diagnostics** menu (`4`) provides health monitoring for your container runtimes and monitoring of your OpenSearch instance.
- **Self-managed containers**: Get container logs with `docker compose logs` or `podman logs`.
- **Docling**: See Stop, start, and inspect native services.

Stop and start containers

- **TUI**: In the TUI's **Status** menu (`3`), click **Stop Services** to stop all OpenRAG container-based services.

  Click **Start All Services** to restart the OpenRAG containers. This function triggers the following processes:

  i. 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 because there are separate Docker Compose files for GPU and CPU deployments.

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

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

- **Self-managed containers**: Use `docker compose down` and `docker compose up -d`.

  To stop or start individual containers, use targeted commands like `docker stop CONTAINER_ID` and `docker start CONTAINER_ID`.

Stop, start, and inspect native services (Docling)

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.

- **TUI**: From the TUI's **Status** menu (`3`), click **Native Services** to do the following:
  - View the service's status, port, and process ID (PID).
  - Stop, start, and restart native services.
- **Self-managed services**: Because the Docling service doesn't run in a container, you must start and stop it manually on the host machine:
  - Stop `docling serve`:

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

  - Start `docling serve`:

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

  - Check that `docling serve` is running:

    ```
    
    ```

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

If `docling serve` is running, the output includes the status, address, and process ID (PID):

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

Upgrade services

See Upgrade OpenRAG.

Reset containers (destructive)

Reset your OpenRAG deployment by recreating the containers and removing some related data.

To completely reset your OpenRAG deployment and delete all OpenRAG data, see Reinstall OpenRAG.

Export customized flows before resetting containers

If you modified the built-in flows or created custom flows in your OpenRAG Langflow instance, and you want to preserve those changes, export your flows before resetting your OpenRAG containers.

Factory Reset with the TUI

> **WARNING**
>
> This is a destructive action that does the following:
>
> - Destroys all OpenRAG containers, volumes, and local images with `docker compose down --volumes --remove-orphans --rmi local`.
> - Prunes any additional Docker objects with `docker system prune -f`.

- Deletes the contents of OpenRAG's `config` and `./opensearch-data` directories.
- Deletes the `conversations.json` file.

Destroyed containers and deleted data are lost and cannot be recovered after running this operation.

This operation *doesn't* remove the `.env` file or the contents of the `./openrag-documents` directory.

1. To destroy and recreate your OpenRAG containers, open the TUI's **Status** menu (`3`), and then click **Factory Reset**.

2. Repeat the setup process to restart the services and launch the OpenRAG app. Your OpenRAG passwords, OAuth credentials (if previously set), and onboarding configuration are restored from the `.env` file.

Rebuild self-managed containers

This command destroys and recreates the containers. Data stored exclusively on the containers is lost, such as Langflow flows.

If you want to preserve customized flows, see Export customized flows before resetting containers.

The `.env` file, `config` directory, `./openrag-documents` directory, `./opensearch-data` directory, and the `conversations.json` file are preserved.

Docker

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

Podman

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

Destroy and recreate self-managed containers

Use separate commands to destroy and recreate the containers if you want to modify the configuration or delete other OpenRAG data before recreating the containers.

> **WARNING**
> These are destructive operations that reset your OpenRAG deployment to an initial state. Destroyed containers and deleted data are lost and cannot be recovered after running this operation.

1. Destroy the containers, volumes, and local images, and then remove (prune) any additional Docker objects:

   Docker

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

   Podman

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

2. Optional: Remove data that wasn't deleted by the previous commands:

   - OpenRAG's `.env` file
   - The contents of OpenRAG's `config` directory
   - The contents of the `./openrag-documents` directory
   - The contents of the `./opensearch-data` directory
   - The `conversations.json` file

3. If you deleted the `.env` file, prepare a new `.env` before redeploying the containers. For more information, see Deploy OpenRAG with self-managed services.

4. Recreate the containers:

   Docker

```
docker compose up -d
```

Podman

```
podman compose up -d
```

5. Launch the OpenRAG app, and then repeat application onboarding.

See also

- Uninstall OpenRAG

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.

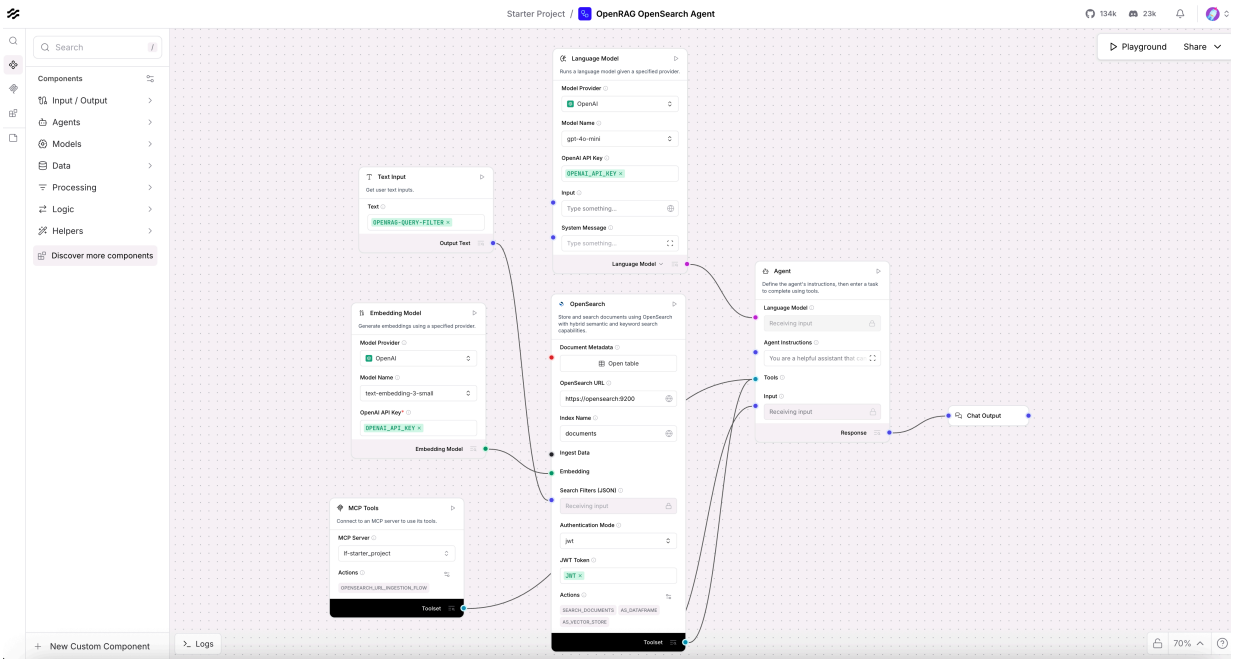
For example, to view and edit the built-in **Chat** flow (the **OpenRAG OpenSearch Agent** flow), do the following:

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.

Default documents

By default, OpenRAG includes some initial documents about OpenRAG. These documents are ingested automatically during application onboarding.

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 provide the initial configuration values for your OpenRAG services. This includes authentication credentials for OpenSearch and OAuth connectors. This configuration determines how OpenRAG authenticates with OpenSearch and controls access to documents in your knowledge base:

- **No-auth mode (basic setup)**: If you select **Basic Setup** in the TUI, or your `.env` file doesn't include OAuth credentials, then the OpenRAG OpenSearch instance runs 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 knowledge base.

- **OAuth mode (advanced setup)**: If you select **Advanced Setup** in the TUI, or your `.env` file includes OAuth credentials, then the OpenRAG OpenSearch instance runs 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; users see only the documents that they uploaded or have access to through their cloud storage accounts.

  You can enable OAuth mode after installation, as explained in 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. For information about starting and stopping OpenRAG native services, like Docling, see Manage OpenRAG 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.

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-managed services

If you use the Terminal User Interface (TUI) to manage your OpenRAG services, enter OAuth credentials in the **Advanced Setup** menu. You can do this during installation, or you can add the credentials afterwards:

1. If OpenRAG is running, open the TUI's **Status** menu (`3`), and then click **Stop Services**.

2. Open the **Advanced Setup** menu (`2`), 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 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** to add the OAuth credentials to your OpenRAG `.env` file.

5. Click **Start All Services** to restart the OpenRAG containers with OAuth enabled.

6. Launch the OpenRAG app. You should be prompted to sign in to your OAuth provider before being redirected to your OpenRAG instance.

Self-managed services

If you installed OpenRAG with self-managed services, 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 all OpenRAG containers:

Docker

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

Podman

```
podman stop --all
```

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 containers:

Docker

```
docker compose up -d
```

Podman

```
podman 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**.

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.

Built-in filters

When you install OpenRAG, it automatically creates an **OpenRAG docs** filter that includes OpenRAG's default documents. These documents provide information about OpenRAG itself and help you learn how to use OpenRAG.

When you use the OpenRAG **Chat**, apply the **OpenRAG docs** filter if you want to ask questions about OpenRAG's features and functionality. This limits the agent's context to the default OpenRAG documentation rather than all documents in your knowledge base.

After uploading your own documents, it is recommended that you create your own filters to organize your documents effectively and separate them from the default OpenRAG documents.

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.

The OpenRAG **Chat** uses an LLM-powered agent to understand your queries, retrieve relevant information from your knowledge base, and generate context-aware responses. The agent can also fetch information from URLs and new documents that you provide during the chat session. To limit the knowledge available to the agent, use filters.

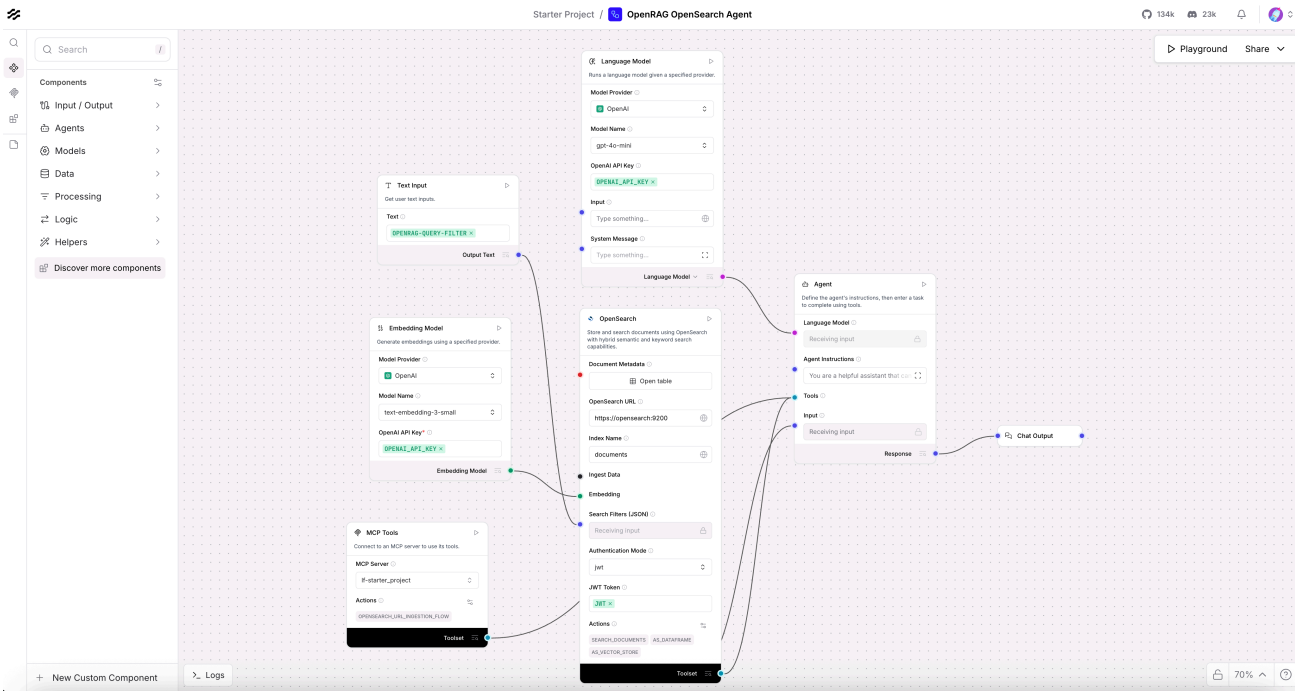
The agent can call specialized Model Context Protocol (MCP) tools to extend its capabilities. To add or change the available tools, you must edit the **OpenRAG OpenSearch Agent** flow.

> **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. When you inspect this flow, you can edit the components to customize the agent's behavior.



- **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.

  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 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. Optional: If you don't want to use the Langflow API key that is generated automatically when you install OpenRAG, you can create a Langflow API key. This key doesn't grant access to OpenRAG; it is only for authenticating with the Langflow API.
    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

```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

Environment variables are either mutable or immutable.

If you edit mutable environment variables, you can apply the changes by stopping and restarting the OpenRAG services after editing the `.env` file:

1. Stop the OpenRAG services.
2. Edit your `.env` file.
3. Restart the OpenRAG services.

If you edit immutable environment variables, you must redeploy OpenRAG with your modified `.env` file. For example, with self-managed services, do the following:

1. Stop the deployment:

Docker

```
docker compose down
```

Podman

```
podman compose down
```

2. Edit your `.env` file.
3. Redeploy OpenRAG:

Docker

```
docker compose up -d
```

Podman

```
podman compose up -d
```

4. Restart the Docling service.
5. Launch the OpenRAG app, and then repeat application onboarding. The values in your `.env` file are automatically populated.

Supported environment variables

All OpenRAG configuration can be controlled through environment variables.

Model provider settings

Configure which models and providers OpenRAG uses to generate text and embeddings. You only need to provide credentials for the providers you are using in OpenRAG.

These variables are initially set during application onboarding. Some of these variables are immutable and can only be changed by redeploying OpenRAG, 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, as one of `openai`, `watsonx`, `ollama`, or `anthropic`.
`ANTHROPIC_API_KEY`	Not set	API key for the Anthropic language model provider.
`OPENAI_API_KEY`	Not set	API key for the OpenAI model provider, which is also the default model provider.
`OLLAMA_ENDPOINT`	Not set	Custom provider endpoint for the Ollama model provider.
`WATSONX_API_KEY`	Not set	API key for the IBM watsonx.ai model provider.
`WATSONX_ENDPOINT`	Not set	Custom provider endpoint for the IBM watsonx.ai model provider.
`WATSONX_PROJECT_ID`	Not set	Project ID for the IBM watsonx.ai model provider.

Document processing settings

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.

Variable	Default	Description
`DISABLE_INGEST_WITH_LANGFLOW`	`false`	Disable Langflow ingestion pipeline.
`DOCLING_OCR_ENGINE`	Set by OS	OCR engine for document processing. For macOS, `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 the OpenRAG Langflow server's authentication, contact point, and built-in flow definitions.

> **INFO**
>
> The `LANGFLOW_SUPERUSER_PASSWORD` is set in your `.env` file, and this value determines the default values for several other Langflow authentication variables.
>
> If the `LANGFLOW_SUPERUSER_PASSWORD` variable isn't set, then the Langflow server starts *without* authentication enabled.
>
> For better security, it is recommended to set `LANGFLOW_SUPERUSER_PASSWORD` so the Langflow server starts with authentication enabled.

Variable	Default	
`LANGFLOW_AUTO_LOGIN`	Determined by `LANGFLOW_SUPERUSER_PASSWORD`	Whether for the L CLI. If `LANGFL` isn't set, `LANGFL`

Variable	Default	
		and auto LANGFL set, ther False a disabled require a Langflov auto-log
LANGFLOW_ENABLE_SUPERUSER_CLI	Determined by LANGFLOW_SUPERUSER_PASSWORD	Whether langfl LANGFL isn't set, LANGFL is True be creat LANGFL set, ther LANGFL is False superus
LANGFLOW_NEW_USER_IS_ACTIVE	Determined by LANGFLOW_SUPERUSER_PASSWORD	Whether account LANGFL isn't set, LANGFL True ar active by LANGFL set, ther LANGFL False a inactive
LANGFLOW_PUBLIC_URL	http://localhost:7860	Public U instance Langflov interface Langflov
LANGFLOW_KEY	Automatically generated	A Langfl Langflov Langflov specific, this key addition deployin

Variable	Default	
`LANGFLOW_SECRET_KEY`	Automatically generated	Secret e internal recomm Langflov If not se secret k
`LANGFLOW_SUPERUSER`	`admin`	Usernam administ
`LANGFLOW_SUPERUSER_PASSWORD`	Not set	Langflov not set, *without* recomm `LANGFL` so the L authenti
`LANGFLOW_URL`	`http://localhost:7860`	URL for
`LANGFLOW_CHAT_FLOW_ID`, `LANGFLOW_INGEST_FLOW_ID`, `NUDGES_FLOW_ID`	Built-in flow IDs	These va to the ID nudges found in change replace custom present OpenRA you depl you can local clo reposito OpenRA
`SYSTEM_PROMPT`	`You are a helpful AI assistant with access to a knowledge base. Answer questions based on the provided context.`	System agent dr

OAuth provider settings

Configure OAuth providers and external service integrations.

Variable	Default	Description
`AWS_ACCESS_KEY_ID` `AWS_SECRET_ACCESS_KEY`	Not set	Enable access to AWS S3 with an AWS OAuth app integration.
`GOOGLE_OAUTH_CLIENT_ID` `GOOGLE_OAUTH_CLIENT_SECRET`	Not set	Enable the Google OAuth client integration. You can generate these values in the Google Cloud Console.
`MICROSOFT_GRAPH_OAUTH_CLIENT_ID` `MICROSOFT_GRAPH_OAUTH_CLIENT_SECRET`	Not set	Enable the Microsoft Graph OAuth client integration by providing Azure application registration credentials for SharePoint and OneDrive.
`WEBHOOK_BASE_URL`	Not set	Base URL for OAuth connector webhook endpoints. If not set, a default base URL is used.

OpenSearch settings

Configure OpenSearch database authentication.

Variable	Default	Description
`OPENSEARCH_HOST`	`localhost`	OpenSearch instance host.
`OPENSEARCH_PORT`	`9200`	OpenSearch instance port.
`OPENSEARCH_USERNAME`	`admin`	OpenSearch administrator username.
`OPENSEARCH_PASSWORD`	Must be set at start up	Required. OpenSearch administrator password. Must adhere to the OpenSearch password complexity requirements. You must set this directly in the `.env` or in the TUI's [**Basic/Advanced Setup**(/install#setup)].

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`	Not set	Set to `json` to enabled JSON-formatted log output. If not set, the default format is used.
`LOG_LEVEL`	`INFO`	Logging level. Can be one of `DEBUG`, `INFO`, `WARNING`, or `ERROR`. `DEBUG` provides the most detailed logs but can impact performance.
`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

With the default configuration, OpenRAG requires the following ports to be available on the host machine:

- 3000: Langflow application
- 5001: Docling local ingestion service
- 5601: OpenSearch Dashboards
- 7860: Docling UI
- 8000: Docling API
- 9200: OpenSearch service

OCR ingestion fails (easyocr not installed)

Docling ingestion can fail with an OCR-related error that mentions `easyocr` is missing. This is likely due to a stale `uv` cache when you install OpenRAG with `uvx`.

When you invoke OpenRAG with `uvx openrag`, `uvx` creates a cached, ephemeral environment that doesn't modify your project. The location and path of this cache depends on your operating system. For example, on macOS, this is typically a user cache directory, such as `/Users/USER_NAME/.cache/uv`.

This cache can become stale, producing errors like missing dependencies.

1. Exit the TUI.

2. Clear the `uv` cache:

   ```
   uv cache clean
   ```

   To clear the OpenRAG cache only, run:

   ```
   uv cache clean openrag
   ```

3. Invoke OpenRAG to restart the TUI:

   ```
   uvx openrag
   ```

4. Click **Open App**, and then retry document ingestion.

If you install OpenRAG with `uv`, dependencies are synced directly from your `pyproject.toml` file. This should automatically install `easyocr` because `easyocr` is included as a dependency in OpenRAG's `pyproject.toml`.

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:

1. Remove only the Langflow container:

   i. Stop the Langflow container:

   Docker

```
docker stop langflow
```

Podman

```
podman stop langflow
```

ii. Remove the Langflow container:

Docker

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

Podman

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

2. Retry the upgrade.
3. If reinstalling the Langflow container doesn't resolve the issue, then you must reset all containers or reinstall OpenRAG.
4. Retry the upgrade.

   If no updates are available after reinstalling OpenRAG, then you reinstalled at the latest version, and your deployment is up to date.

Document ingestion or similarity search issues

See Troubleshoot ingestion.