What is OpenRAG?

OpenRAG is an open-source package for building agentic RAG systems that integrates with a wide range of orchestration tools, 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, popular 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 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-authorized 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 document 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-authorized 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

- Install Python version 3.12 or later.
- Install uv.
- Get an OpenAI API key. This quickstart uses OpenAI for demonstration purposes. For other providers, see the other installation methods.
- For Microsoft Windows, you must use the Windows Subsystem for Linux (WSL). See Install OpenRAG on Windows before proceeding.

Install OpenRAG

This quickstart installs OpenRAG with `uvx` which automatically installs OpenRAG's dependencies as needed, including Docker or Podman.

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

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

2. Install OpenRAG and its dependencies with `uvx`.

   ```
   uvx 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.

If you encounter errors during installation, see Troubleshoot OpenRAG.

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

4. For **Langflow Admin Password**, click **Generate Password** to create a Langflow administrator password and username.

5. For all other fields, use the default values or leave them empty.

6. Click **Save Configuration**.

   Your OpenRAG configuration and passwords are stored in an OpenRAG `.env` file file that is created automatically at `~/.openrag/tui`. OpenRAG container definitions are stored in the `docker-compose` files in the same directory.

7. Click **Start OpenRAG** to start the OpenRAG 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
   ```

8. Click **Close**, and then click **Launch OpenRAG** to access 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 onboarding chat 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 knowledge base. You can click a document to view the chunks of the document as they are stored in the knowledge base.

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 `~/.openrag/documents`.

   For other ingestion options, see Ingest knowledge.

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

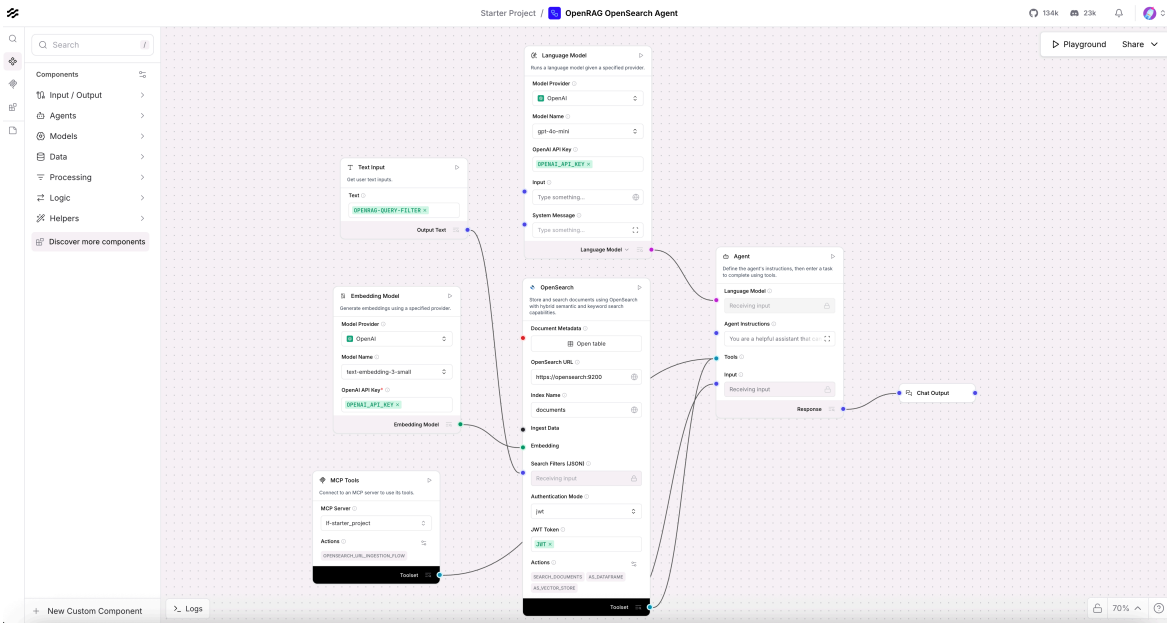
If the agent's response doesn't seem to reference your documents correctly, see Troubleshoot chat issues.

Change the language model and chat settings

In OpenRAG, you can change some chat and ingestion settings directly on the **Settings** page. For full customization capabilities, you can edit the underlying Langflow flows in the OpenRAG's embedded Langflow visual editor.

For this quickstart, try changing the **Language Model** or **Agent Instructions**. Editing the flow in Langflow is optional.

1. In OpenRAG, click **Settings**, and then find the **Agent** section.
2. If you only need to edit the language model or agent instructions, edit those fields directly on the **Settings** page. Language model changes are saved automatically. To apply new instructions, click **Save Agent Instructions**.
3. To edit all flow settings and components with full customization capabilities, edit the flow in the Langflow visual editor:
    i. Click **Edit in Langflow** to launch the Langflow visual editor in a new browser tab.
    ii. If prompted to acknowledge that you are entering Langflow, click **Proceed**.
    iii. If Langflow requests login information, enter the `LANGFLOW_SUPERUSER` and `LANGFLOW_SUPERUSER_PASSWORD` from your OpenRAG `.env` file.
    iv. In the Langflow header, click **Starter Project** to go to the Langflow **Projects** page, and then unlock the flow.
    v. On the **Projects** page, click the **OpenRAG OpenSearch Agent** flow to open the visual editor with the flow unlocked.



vi. Modify the flow as desired, and then press `Command`+`S` (`Ctrl`+`S`) to save your changes. Then, you can close the Langflow browser tab, or leave it open if you want to continue experimenting with the flow editor.

4. After you modify any **Agent** flow settings, go to the OpenRAG **Chat**, and then click **Start new conversation** in the **Conversations** list. This ensures that the chat doesn't persist any context from the previous conversation with the original flow settings.

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

6. To undo your changes, go to **Settings**, and then click **Restore flow** to revert the flow to its original state when you first installed OpenRAG.

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.

- **Use the OpenRAG SDKs**: To interact with OpenRAG programmatically, use the OpenRAG SDKs.

- **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. Use one of the following options:
    - `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. This option also automatically installs Docker or Podman if neither is present in your environment.
- **Install OpenRAG on Microsoft Windows**: On Windows machines, you must install OpenRAG within the Windows Subsystem for Linux (WSL).

    > **WARNING**
    > 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 the application onboarding process. 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. When using TUI-managed services, this file is created automatically at `~/.openrag/tui`, or you can provide a pre-populated `.env` file in this directory before starting the TUI. The TUI prompts you for the required values during setup and onboarding, and any values detected in a preexisting

`.env` file are populated automatically. When using self-managed services, you must provide a pre-populated `.env` file, as you would for any Docker or Podman deployment. For more information, see the instructions for your preferred installation method and the OpenRAG environment variables reference.

Install OpenRAG in a Python project with uv

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

When you install OpenRAG with `uv`, you will use the Terminal User Interface (TUI) to configure and manage your OpenRAG deployment.

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.
- Install Python version 3.12 or later.
- Install uv.
- Install Podman or Docker.

  The OpenRAG team recommends, at minimum, 8 GB of RAM for container VMs. However, if you plan to upload large files regularly, more RAM is recommended. For more information, see Troubleshoot OpenRAG.

  A Docker or Podman VM must be running before you start OpenRAG.
- Install `podman-compose` or Docker Compose. To use Docker Compose with Podman, you must alias Docker Compose commands to Podman commands.
- Gather the credentials and connection details for one or more supported model 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. Then, get your Ollama server's base URL and the names of the

models that you want to use.

OpenRAG requires at least one language model and one embedding model. If a provider offers both types of models, then you can use the same provider for both models. If a provider offers only one type, then you must configure two providers.

Language models must support tool calling to be compatible with OpenRAG.

For more information, see Complete the application onboarding process.

- 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 that is suitable for most use cases. The default CPU-only deployment doesn't prevent you from using GPU acceleration in external services, such as Ollama servers.

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.

Use uv add

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

```
uv init PROJECT_NAME --python 3.12
```

The `--python` flag ensures that your project uses the minimum required Python version for OpenRAG. You can use a later Python version, if you prefer. Or, you can omit this flag if your system's default Python version is 3.12 or later.

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. Optional: If you want to use a pre-populated OpenRAG `.env` file, create one at `~/.openrag/tui` before starting OpenRAG.

5. Start the OpenRAG TUI:

```
uv run openrag
```

Use uv pip install

1. Activate your virtual environment.

2. Install the OpenRAG Python package:

```
uv pip install openrag
```

3. Optional: If you want to use a pre-populated OpenRAG `.env` file, create one at `~/.openrag/tui` before starting OpenRAG.

4. 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 TUI. The TUI guides you through the initial configuration process before you start the OpenRAG services.

Your configuration values are stored in an OpenRAG `.env` file that is created automatically at `~/.openrag/tui`. If OpenRAG detects an existing `.env` file in this directory, then the TUI can populate those values automatically during setup and onboarding.

Container definitions are stored in the `docker-compose` files in the same directory as the OpenRAG `.env` file.

1. In the TUI, click either **Basic Setup** or **Advanced Setup**.

> **INFO**
>
> You must use **Advanced Setup** if you want to enable either OAuth mode or cloud storage connectors during initial set up:
>
> - **OAuth mode**: Controls document ownership and access in your OpenRAG OpenSearch knowledge base. Without OAuth mode, there is no differentiation between users; all users that access your OpenRAG instance can access and manage all uploaded documents.
> - **Cloud storage connectors**: Enables ingestion of documents from external storage services.

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

You can also enable these features later.

2. Enter administrator passwords for the OpenRAG OpenSearch and Langflow services.

   The OpenSearch password is required, and a secure password is automatically generated if you don't provide one manually.

   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.

   You can click **Generate Password** to create a Langflow password and username automatically.

3. Optional: Under **API Keys**, enter your model provider credentials, or leave these fields empty if you want to configure model provider credentials during the application onboarding process.

   There is no material difference between providing these values now or during the application onboarding process. If you provide a credential now, it can be populated automatically during the application onboarding process if you enable the **Use environment API key** option.

   OpenRAG's core functionality requires access to language and embedding models. By default, OpenRAG uses OpenAI models. If you aren't sure which models or providers to use, you must provide an OpenAI API key to use OpenRAG's default model configuration.

4. **Advanced Setup** only: To enable OAuth mode or cloud storage connectors, do the following:

   i. 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.

ii. Enter the relevant OAuth credentials under **API Keys** in the TUI's **Advanced Setup**:

- **Google**: Enter 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.

  Providing these Google credentials enables OAuth mode *and* the Google Drive cloud storage connector.

  > **WARNING**
  > Google is the only supported OAuth provider for OpenRAG.
  >
  > You must enter Google credentials if you want to enable OAuth mode.
  >
  > The Microsoft and Amazon credentials are used only to authorize the cloud storage connectors. OpenRAG doesn't offer OAuth provider integrations for Microsoft or Amazon.

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

- **Amazon**: Enter 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.

The credentials can be populated automatically if OpenRAG detects these credentials in an OpenRAG `.env` file at `~/.openrag/tui`.

iii. Register the redirect URIs shown in the TUI in your OAuth apps.

The redirect URIs are used for the cloud storage connector webhooks. For Google, the redirect URIs are also used to redirect users back to OpenRAG after they sign in.

5. Optional: Under **Langfuse (Tracing)**, enter Langfuse credentials to enable the Langflow integration with Langfuse:

- **Langfuse Secret Key**: A secret key for your Langfuse project.
- **Langfuse Public Key**: A public key for your Langfuse project.
- **Langfuse Host**: Required for self-hosted Langfuse deployments. Leave empty for Langfuse Cloud.

6. Optional: Under **Others**, you can edit the following settings if needed:

    - **Documents Paths**: Set the local documents path. Use the default path, or provide a path to a directory where you want OpenRAG to look for documents to ingest into your knowledge base.
    - **OpenSearch Data Path**: Use the default path, or specify the path where you want OpenRAG to create your OpenSearch index.
    - **Langflow Public URL (Advanced Setup only)** : Sets the base address to access the Langflow web interface where users interact with the Langflow editor in a browser. You must set this value to run Langflow on a non-default port (`7860`)
    - **Webhook Base URL (Advanced Setup only)**: If you entered OAuth credentials, you can set the base address for your OAuth connector endpoints. If set, the OAuth connector webhook URLs are constructed as `WEBHOOK_BASE_URL/connectors/${provider}/webhook`. This option is required to enable automatic ingestion from cloud storage.

7. Click **Save Configuration**.

    Your passwords, API key, and OAuth credentials (if provided) are stored in the OpenRAG `.env` file at `~/.openrag/tui`. If you modified any credentials that were pulled from an existing `.env` file, those values are updated in the `.env` file.

8. Click **Start OpenRAG** to start the OpenRAG 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
    ```

9. Click **Close**, and then click **Launch OpenRAG** or navigate to `localhost:3000` in your browser.

If you provided Google OAuth credentials, you must sign in with Google before you are redirected to your OpenRAG instance.

10. Continue with the application onboarding process.

Complete the application onboarding process

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

To complete onboarding, you must configure at least one language model and one embedding model.

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 select multiple embedding models.

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 **Use environment API key** to pull the key from your OpenRAG `.env` file.
2. Under **Advanced settings**, select the language model that you want to use.

   Language models must support tool calling to be compatible with OpenRAG. Incompatible models aren't listed.
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 the application onboarding screen. 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

> **INFO**
>
> OpenRAG isn't guaranteed to be compatible with all models that are available through IBM watsonx.ai.
>
> Language models must support tool calling to be compatible with OpenRAG. Incompatible models aren't listed in OpenRAG's settings or onboarding.
>
> Additionally, models must be able to handle the agentic reasoning tasks required by OpenRAG. Models that are too small or not designed for agentic RAG tasks can return low quality, incorrect, or improperly formatted responses. For more information, see Chat issues.
>
> You can submit an OpenRAG GitHub issue to request support for specific models.

1. For **watsonx.ai API Endpoint**, select the base URL for your watsonx.ai model deployment.

2. Enter your watsonx.ai deployment's project ID and API key.

   You can enable **Use environment API key** to pull the key from your OpenRAG `.env` file.

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

   Language models must support tool calling to be compatible with OpenRAG. Incompatible models aren't listed.

4. Click **Complete**.

5. 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.

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

7. 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

Using Ollama as your language and embedding model provider offers greater flexibility and configuration options for hosting models. However, it requires additional setup because Ollama isn't included with OpenRAG. You must deploy Ollama separately if you want to use Ollama as a model provider.

> **INFO**
>
> OpenRAG isn't guaranteed to be compatible with all models that are available through Ollama. Some models might produce unexpected results, such as JSON-formatted output instead of natural language responses, and some models aren't appropriate for the types of tasks that OpenRAG performs, such as those that generate media.
>
> - **Language models**: Ollama-hosted language models must support tool calling to be compatible with OpenRAG. The OpenRAG team recommends `gpt-oss:20b` or `mistral-nemo:12b`. If you choose `gpt-oss:20b`, consider using Ollama Cloud or running Ollama on a remote machine because this model requires at least 16GB of RAM.

- **Embedding models**: The OpenRAG team recommends `nomic-embed-text:latest`, `mxbai-embed-large:latest`, or `embeddinggemma:latest`.

You can experiment with other models, but if you encounter issues that you are unable to resolve through other RAG best practices (like context filters and prompt engineering), try switching to one of the recommended models. You can submit an OpenRAG GitHub issue to request support for specific models.

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 the OpenRAG onboarding dialog, enter your Ollama server's base URL:

   - **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`.

3. Select a language model that your Ollama server is running.

   OpenRAG only lists language models that support tool calling. If your server isn't running any compatible language models, you must either deploy a compatible language model on your Ollama server, or use another provider for the language model.

   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**.

5. 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.

6. 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 the application onboarding screen. Verify that the server address is valid, and that the selected model is running on the server. Then, click **Complete** to retry ingestion.

7. 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

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

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

   Language models must support tool calling to be compatible with OpenRAG. Incompatible models aren't listed.

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 the application

onboarding screen. 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

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

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.

When you install OpenRAG with `uvx`, you will use the Terminal User Interface (TUI) to configure and manage your OpenRAG deployment.

> **INFO**
>
> `uvx openrag` installs the latest version of OpenRAG and dependencies, like Docker or Podman, if they aren't already present in your environment.

Prerequisites

- For Microsoft Windows, you must use the Windows Subsystem for Linux (WSL). See Install OpenRAG on Windows before proceeding.
- Install Python version 3.12 or later.
- Install uv.
- Gather the credentials and connection details for one or more supported model 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. Then, get your Ollama server's base URL and the names of the models that you want to use.

  OpenRAG requires at least one language model and one embedding model. If a provider offers both types of models, then you can use the same provider for both models. If a provider offers only one type, then you must configure two providers.

Language models must support tool calling to be compatible with OpenRAG.

For more information, see Complete the application onboarding process.

- 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 that is suitable for most use cases. The default CPU-only deployment doesn't prevent you from using GPU acceleration in external services, such as Ollama servers.

Install and run OpenRAG with uvx

1. Create a directory for 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 OpenRAG `.env` file, create one at `~/.openrag/tui` before invoking OpenRAG.

3. Invoke OpenRAG and install any missing dependencies:

```
uvx openrag
```

You can invoke a specific version using any of the `uvx` version specifiers, such as `uvx --from openrag==0.1.30 openrag`.

4. If needed, respond to any prompts to install and configure dependencies like Docker or Podman.

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

Invoking OpenRAG with `uvx openrag` creates a cached, ephemeral environment for the TUI in your local `uv` cache. Your OpenRAG configuration files and data are stored in `~/.openrag`, which is separate from the `uv` cache. Clearing the `uv` cache doesn't

remove your entire OpenRAG installation; it only clears `uv` dependencies. 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 TUI. The TUI guides you through the initial configuration process before you start the OpenRAG services.

Your configuration values are stored in an OpenRAG `.env` file that is created automatically at `~/.openrag/tui`. If OpenRAG detects an existing `.env` file in this directory, then the TUI can populate those values automatically during setup and onboarding.

Container definitions are stored in the `docker-compose` files in the same directory as the OpenRAG `.env` file.

1. In the TUI, click either **Basic Setup** or **Advanced Setup**.

   > **INFO**
   >
   > You must use **Advanced Setup** if you want to enable either OAuth mode or cloud storage connectors during initial set up:
   >
   > - **OAuth mode**: Controls document ownership and access in your OpenRAG OpenSearch knowledge base. Without OAuth mode, there is no differentiation between users; all users that access your OpenRAG instance can access and manage all uploaded documents.
   > - **Cloud storage connectors**: Enables ingestion of documents from external storage services.

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

   You can also enable these features later.

2. Enter administrator passwords for the OpenRAG OpenSearch and Langflow services.

   The OpenSearch password is required, and a secure password is automatically generated if you don't provide one manually.

   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.

   You can click **Generate Password** to create a Langflow password and username automatically.

3. Optional: Under **API Keys**, enter your model provider credentials, or leave these fields empty if you want to configure model provider credentials during the application onboarding process.

   There is no material difference between providing these values now or during the application onboarding process. If you provide a credential now, it can be populated automatically during the application onboarding process if you enable the **Use environment API key** option.

   OpenRAG's core functionality requires access to language and embedding models. By default, OpenRAG uses OpenAI models. If you aren't sure which models or providers to use, you must provide an OpenAI API key to use OpenRAG's default model configuration.

4. **Advanced Setup** only: To enable OAuth mode or cloud storage connectors, do the following:

   i. 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.

   ii. Enter the relevant OAuth credentials under **API Keys** in the TUI's **Advanced Setup**:

      - **Google**: Enter 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.

Providing these Google credentials enables OAuth mode *and* the Google Drive cloud storage connector.

> **WARNING**
>
> Google is the only supported OAuth provider for OpenRAG.
>
> You must enter Google credentials if you want to enable OAuth mode.
>
> The Microsoft and Amazon credentials are used only to authorize the cloud storage connectors. OpenRAG doesn't offer OAuth provider integrations for Microsoft or Amazon.

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

- **Amazon**: Enter 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.

The credentials can be populated automatically if OpenRAG detects these credentials in an OpenRAG `.env` file at `~/.openrag/tui`.

iii. Register the redirect URIs shown in the TUI in your OAuth apps.

The redirect URIs are used for the cloud storage connector webhooks. For Google, the redirect URIs are also used to redirect users back to OpenRAG after they sign in.

5. Optional: Under **Langfuse (Tracing)**, enter Langfuse credentials to enable the Langflow integration with Langfuse:

   - **Langfuse Secret Key**: A secret key for your Langfuse project.
   - **Langfuse Public Key**: A public key for your Langfuse project.
   - **Langfuse Host**: Required for self-hosted Langfuse deployments. Leave empty for Langfuse Cloud.

6. Optional: Under **Others**, you can edit the following settings if needed:

- **Documents Paths**: Set the local documents path. Use the default path, or provide a path to a directory where you want OpenRAG to look for documents to ingest into your knowledge base.
- **OpenSearch Data Path**: Use the default path, or specify the path where you want OpenRAG to create your OpenSearch index.
- **Langflow Public URL (Advanced Setup only)** : Sets the base address to access the Langflow web interface where users interact with the Langflow editor in a browser. You must set this value to run Langflow on a non-default port (`7860`)
- **Webhook Base URL (Advanced Setup only)**: If you entered OAuth credentials, you can set the base address for your OAuth connector endpoints. If set, the OAuth connector webhook URLs are constructed as `WEBHOOK_BASE_URL/connectors/${provider}/webhook`. This option is required to enable automatic ingestion from cloud storage.

7. Click **Save Configuration**.

   Your passwords, API key, and OAuth credentials (if provided) are stored in the OpenRAG `.env` file at `~/.openrag/tui`. If you modified any credentials that were pulled from an existing `.env` file, those values are updated in the `.env` file.

8. Click **Start OpenRAG** to start the OpenRAG 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
   ```

9. Click **Close**, and then click **Launch OpenRAG** or navigate to `localhost:3000` in your browser.

   If you provided Google OAuth credentials, you must sign in with Google before you are redirected to your OpenRAG instance.

10. Continue with the application onboarding process.

Complete the application onboarding process

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

To complete onboarding, you must configure at least one language model and one embedding model.

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 select multiple embedding models.

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 **Use environment API key** to pull the key from your OpenRAG `.env` file.
2. Under **Advanced settings**, select the language model that you want to use.

   Language models must support tool calling to be compatible with OpenRAG. Incompatible models aren't listed.
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 the application

onboarding screen. 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

> **INFO**
>
> OpenRAG isn't guaranteed to be compatible with all models that are available through IBM watsonx.ai.
>
> Language models must support tool calling to be compatible with OpenRAG. Incompatible models aren't listed in OpenRAG's settings or onboarding.
>
> Additionally, models must be able to handle the agentic reasoning tasks required by OpenRAG. Models that are too small or not designed for agentic RAG tasks can return low quality, incorrect, or improperly formatted responses. For more information, see Chat issues.
>
> You can submit an OpenRAG GitHub issue to request support for specific models.

1. For **watsonx.ai API Endpoint**, select the base URL for your watsonx.ai model deployment.

2. Enter your watsonx.ai deployment's project ID and API key.

   You can enable **Use environment API key** to pull the key from your OpenRAG `.env` file.

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

   Language models must support tool calling to be compatible with OpenRAG. Incompatible models aren't listed.

4. Click **Complete**.

5. 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.

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

7. 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

Using Ollama as your language and embedding model provider offers greater flexibility and configuration options for hosting models. However, it requires additional setup because Ollama isn't included with OpenRAG. You must deploy Ollama separately if you want to use Ollama as a model provider.

> **INFO**
>
> OpenRAG isn't guaranteed to be compatible with all models that are available through Ollama. Some models might produce unexpected results, such as JSON-formatted output instead of natural language responses, and some models aren't appropriate for the types of tasks that OpenRAG performs, such as those that generate media.
>
> - **Language models**: Ollama-hosted language models must support tool calling to be compatible with OpenRAG. The OpenRAG team recommends `gpt-oss:20b` or `mistral-nemo:12b`. If you choose `gpt-oss:20b`, consider using Ollama Cloud or running Ollama on a remote machine because this model requires at least 16GB of RAM.

- **Embedding models**: The OpenRAG team recommends `nomic-embed-text:latest`, `mxbai-embed-large:latest`, or `embeddinggemma:latest`.

You can experiment with other models, but if you encounter issues that you are unable to resolve through other RAG best practices (like context filters and prompt engineering), try switching to one of the recommended models. You can submit an OpenRAG GitHub issue to request support for specific models.

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 the OpenRAG onboarding dialog, enter your Ollama server's base URL:

   - **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`.

3. Select a language model that your Ollama server is running.

   OpenRAG only lists language models that support tool calling. If your server isn't running any compatible language models, you must either deploy a compatible language model on your Ollama server, or use another provider for the language model.

   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**.

5. 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.

6. 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 the application onboarding screen. Verify that the server address is valid, and that the selected model is running on the server. Then, click **Complete** to retry ingestion.

7. 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

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

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

   Language models must support tool calling to be compatible with OpenRAG. Incompatible models aren't listed.

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 the application

onboarding screen. 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).

> **WARNING**
>
> 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.
- Install Python version 3.12 or later.
- Install uv.
- Install Podman or Docker.

  The OpenRAG team recommends, at minimum, 8 GB of RAM for container VMs. However, if you plan to upload large files regularly, more RAM is recommended. For more information, see Troubleshoot OpenRAG.

  A Docker or Podman VM must be running before you start OpenRAG.
- Install `podman-compose` or Docker Compose. To use Docker Compose with Podman, you must alias Docker Compose commands to Podman commands.
- Gather the credentials and connection details for one or more supported model 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. Then, get your Ollama server's base URL and the names of the models that you want to use.

OpenRAG requires at least one language model and one embedding model. If a provider offers both types of models, then you can use the same provider for both models. If a provider offers only one type, then you must configure two providers.

Language models must support tool calling to be compatible with OpenRAG.

For more information, see Complete the application onboarding process.

- 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 that is suitable for most use cases. The default CPU-only deployment doesn't prevent you from using GPU acceleration in external services, such as Ollama servers.

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. If `LANGFLOW_SUPERUSER` isn't set, then the default value is `admin`.

    - **`LANGFLOW_SUPERUSER_PASSWORD` (Strongly recommended)**: Sets the Langflow administrator password, and determines the Langflow server's default authentication mode. If `LANGFLOW_SUPERUSER_PASSWORD` isn't set, then 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 `LANGFLOW_SECRET_KEY` isn't set, then Langflow generates a secret key automatically.

    - **Model provider credentials**: Provide credentials for your preferred model providers. If none of these are set in the `.env` file, you must configure at least one provider during the application onboarding process.

        - `OPENAI_API_KEY`
        - `ANTHROPIC_API_KEY`
        - `OLLAMA_ENDPOINT`
        - `WATSONX_API_KEY`
        - `WATSONX_ENDPOINT`
        - `WATSONX_PROJECT_ID`

6. To enable OAuth mode or cloud storage connectors, do the following:

    i. 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.

ii. In your `.env` file, set the OAuth environment variables for the providers that you want to use:

```
GOOGLE_OAUTH_CLIENT_ID=
GOOGLE_OAUTH_CLIENT_SECRET=
MICROSOFT_GRAPH_OAUTH_CLIENT_ID=
MICROSOFT_GRAPH_OAUTH_CLIENT_SECRET=
AWS_ACCESS_KEY_ID=
AWS_SECRET_ACCESS_KEY=
```

- **Google**: Enter 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.

  Providing these Google credentials enables OAuth mode *and* the Google Drive cloud storage connector.

  > **WARNING**
  >
  > Google is the only supported OAuth provider for OpenRAG.
  >
  > You must enter Google credentials if you want to enable OAuth mode.
  >
  > The Microsoft and Amazon credentials are used only to authorize the cloud storage connectors. OpenRAG doesn't offer OAuth provider integrations for Microsoft or Amazon.

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

- **Amazon**: Enter 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.

iii. For each connector, you must register the OpenRAG redirect URIs in your OAuth apps:

   - Local deployments: `http://localhost:3000/auth/callback`
   - Production deployments: `https://your-domain.com/auth/callback`

   The redirect URIs are used for the cloud storage connector webhooks. For Google, the redirect URIs are also used to redirect users back to OpenRAG after they sign in.

iv. Optional: Set the `WEBHOOK_BASE_URL` to the base address for your OAuth connector endpoints. If set, the OAuth connector webhook URLs are constructed as `WEBHOOK_BASE_URL/connectors/${provider}/webhook`. This option is required to enable automatic ingestion from cloud storage.

7. Optional: To enable the Langflow integration with Langfuse, set the following variables:

   - **`LANGFUSE_SECRET_KEY`**: A secret key for your Langfuse project.
   - **`LANGFUSE_PUBLIC_KEY`**: A public key for your Langfuse project.
   - **`LANGFUSE_HOST`**: Required for self-hosted Langfuse deployments. Leave empty for Langfuse Cloud.

8. Save your `.env` file.

Start services

1. To use the default Docling Serve implementation, start `docling serve` on port 5001 on the host machine using the included script:

   ```
   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.

   Port 5001 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.

> **TIP**
> If you don't want to use the default Docling Serve implementation, see Select a Docling implementation.

2. Confirm `docling serve` is running.

   The following command checks the status of the default Docling Serve implementation:

   ```
   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 configuration for your environment:

   - **CPU-only deployment (Default and recommended)**: If your host machine doesn't have NVIDIA GPU support, use the base `docker-compose.yml` file:

     Docker

     ```
     docker compose up -d
     ```

     Podman

     ```
     podman compose up -d
     ```

   - **GPU-accelerated deployment**: If your host machine has an NVIDIA GPU with CUDA support and compatible NVIDIA drivers, use the base `docker-`

`compose.yml` file with the `docker-compose.gpu.yml` override:

Docker

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

Podman

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

> **TIP**
>
> GPU acceleration isn't required for most use cases. OpenRAG's CPU-only deployment doesn't prevent you from using GPU acceleration in external services, such as Ollama servers.
>
> GPU acceleration is required only for specific use cases, typically involving customization of the ingestion flows or ingestion logic. For example, writing alternate ingest logic in OpenRAG that uses GPUs directly in the container, or customizing the ingestion flows to use Langflow's Docling component with GPU acceleration instead of OpenRAG's Docling Serve service.

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.
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 the application onboarding process.

   If you provided Google OAuth credentials, you must sign in with Google before you are redirected to your OpenRAG instance.

Complete the application onboarding process

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

To complete onboarding, you must configure at least one language model and one embedding model.

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 select multiple embedding models.

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 **Use environment API key** to pull the key from your OpenRAG `.env` file.
2. Under **Advanced settings**, select the language model that you want to use.

   Language models must support tool calling to be compatible with OpenRAG. Incompatible models aren't listed.
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 the application onboarding screen. 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

> **INFO**
>
> OpenRAG isn't guaranteed to be compatible with all models that are available through IBM watsonx.ai.
>
> Language models must support tool calling to be compatible with OpenRAG. Incompatible models aren't listed in OpenRAG's settings or onboarding.

> Additionally, models must be able to handle the agentic reasoning tasks required by OpenRAG. Models that are too small or not designed for agentic RAG tasks can return low quality, incorrect, or improperly formatted responses. For more information, see Chat issues.
>
> You can submit an OpenRAG GitHub issue to request support for specific models.

1. For **watsonx.ai API Endpoint**, select the base URL for your watsonx.ai model deployment.
2. Enter your watsonx.ai deployment's project ID and API key.

   You can enable **Use environment API key** to pull the key from your OpenRAG `.env` file.
3. Under **Advanced settings**, select the language model that you want to use.

   Language models must support tool calling to be compatible with OpenRAG. Incompatible models aren't listed.
4. Click **Complete**.
5. 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.
6. 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 the application onboarding screen. Verify that the credentials are valid and have access to the selected model, and then click **Complete** to retry ingestion.
7. 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

Using Ollama as your language and embedding model provider offers greater flexibility and configuration options for hosting models. However, it requires additional setup because Ollama isn't included with OpenRAG. You must deploy Ollama separately if you want to use Ollama as a model provider.

> **INFO**
>
> OpenRAG isn't guaranteed to be compatible with all models that are available through Ollama. Some models might produce unexpected results, such as JSON-formatted output instead of natural language responses, and some models aren't appropriate for the types of tasks that OpenRAG performs, such as those that generate media.
>
> - **Language models**: Ollama-hosted language models must support tool calling to be compatible with OpenRAG. The OpenRAG team recommends `gpt-oss:20b` or `mistral-nemo:12b`. If you choose `gpt-oss:20b`, consider using Ollama Cloud or running Ollama on a remote machine because this model requires at least 16GB of RAM.
> - **Embedding models**: The OpenRAG team recommends `nomic-embed-text:latest`, `mxbai-embed-large:latest`, or `embeddinggemma:latest`.
>
> You can experiment with other models, but if you encounter issues that you are unable to resolve through other RAG best practices (like context filters and prompt engineering), try switching to one of the recommended models. You can submit an OpenRAG GitHub issue to request support for specific models.

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 the OpenRAG onboarding dialog, enter your Ollama server's base URL:

   - **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`.

3. Select a language model that your Ollama server is running.

   OpenRAG only lists language models that support tool calling. If your server isn't running any compatible language models, you must either deploy a compatible language model on your Ollama server, or use another provider for the language model.

   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**.

5. 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.

6. 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 the application onboarding screen. Verify that the server address is valid, and that the selected model is running on the server. Then, click **Complete** to retry ingestion.

7. 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

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

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

   Language models must support tool calling to be compatible with OpenRAG. Incompatible models aren't listed.

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 the application onboarding screen. 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 deployments

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 OpenRAG `.env` file.

1. If you modified the built-in flows or created custom flows in your OpenRAG Langflow instance, export your flows before starting this process. Although OpenRAG can preserve changes to the built-in flows, it doesn't preserve user-created flows. As a general best practice, exporting your flows is recommended to create backups of your customizations. Afterwards, you can reimport your flows or reference the exported flow JSON as needed.

2. To check for updates, click **Status** in the TUI, and then click **Upgrade**.

3. If there is an update available, press `Esc` to close the **Status** page, then then click **Stop All Services**.

4. Press `q` to exit the TUI.

5. Upgrade the OpenRAG Python package to the latest version from PyPI. The commands to upgrade the package depend on how you installed OpenRAG.

    - `uvx`:

      Use these steps to upgrade the Python package if you installed OpenRAG using `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
      ```

    - `uv add`:

      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
```

- `uv pip install`:

  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
  ```

6. In the OpenRAG TUI, click **Start Services**, and then wait while the services 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 OpenRAG `.env` file, the `OPENRAG_VERSION` environment variable is set to `latest` by default, which 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.

7. After the containers start, click **Close**, and then click **Launch OpenRAG**.

Upgrade self-managed deployments

1. If you modified the built-in flows or created custom flows in your OpenRAG Langflow instance, export your flows before starting this process. Although OpenRAG can preserve changes to the built-in flows, it doesn't preserve user-created flows. As a general best practice, exporting your flows is recommended to create backups of your customizations. Afterwards, you can reimport your flows or reference the exported flow JSON as needed.

2. Fetch and apply the latest container images while preserving your OpenRAG data:

   Docker

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

   Podman

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

   By default, OpenRAG's `docker-compose` files pull the latest container images.

3. After the containers start, access the OpenRAG application at `http://localhost:3000`.

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, such as 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.

Reinstall TUI-managed containers

1. If you modified the built-in flows or created custom flows in your OpenRAG Langflow instance, export your flows before starting this process. Although OpenRAG can preserve changes to the built-in flows, it doesn't preserve user-created flows. As a general best practice, exporting your flows is recommended to create backups of your customizations. Afterwards, you can reimport your flows or reference the exported flow JSON as needed.

2. In the TUI, click **Status**, and then click **Factory Reset** to reset your OpenRAG containers.

   > **WARNING**
   > This is a destructive operation that does the following:
   >
   > - Destroys all OpenRAG containers, volumes, and local images.
   > - Prunes any additional container objects.
   > - Deletes the contents of the `~/.openrag` directory *except* for OpenRAG's `.env` file and the `/documents` subdirectory.
   >
   > Destroyed containers and deleted data are lost and cannot be recovered after running this operation.

3. Press `Esc` to close the **Status** page, and then press `q` to exit the TUI.

4. Optional: Delete or edit OpenRAG's `.env` file, which is stored at `~/.openrag/tui`.

This file contains your OpenRAG configuration, including OpenRAG passwords, API keys, OAuth credentials, and other environment variables. If you delete this file, the TUI automatically generates a new one after you repeat the setup and onboarding process. If you preserve this file, the TUI can read values from the existing `.env` file during setup and onboarding.

5. Optional: Remove any files from the `~/.openrag/documents` subdirectory that you don't want to reingest after redeploying the containers. It is recommended that you preserve OpenRAG's default documents.

6. Restart the TUI with `uv run openrag` or `uvx openrag`.

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

Reinstall self-managed containers with `docker compose` or `podman compose`

Use these steps to reinstall OpenRAG containers with streamlined `docker compose` or `podman compose` commands:

1. If you modified the built-in flows or created custom flows in your OpenRAG Langflow instance, export your flows before starting this process. Although OpenRAG can preserve changes to the built-in flows, it doesn't preserve user-created flows. As a general best practice, exporting your flows is recommended to create backups of your customizations. Afterwards, you can reimport your flows or reference the exported flow JSON as needed.

2. Destroy the containers, volumes, and local images, and then remove (prune) any additional container objects.

   > **WARNING**
   >
   > This is a destructive operation that does the following:
   >
   > - Destroys all OpenRAG containers, volumes, and local images.
   > - Prunes any additional container objects.
   > - Deletes the contents of the `~/.openrag` directory *except* for OpenRAG's `.env` file and the `/documents` subdirectory.

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

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
```

3. Optional: Edit OpenRAG's `.env` file if needed.
4. Optional: Remove any files from the `~/.openrag/documents` subdirectory that you don't want to reingest after redeploying the containers. It is recommended that you preserve OpenRAG's default documents.
5. Redeploy OpenRAG:

Docker

```
docker compose up -d
```

Podman

```
podman compose up -d
```

6. Launch the OpenRAG app, and then repeat the application onboarding process.

Reinstall self-managed containers with discrete `docker` or `podman` commands

Use these commands to remove and clean up OpenRAG containers with discrete `docker` or `podman` commands.

If you want to reinstall one container, specify the container name in the commands instead of running the commands on all containers.

1. If you modified the built-in flows or created custom flows in your OpenRAG Langflow instance, export your flows before starting this process. Although OpenRAG can preserve changes to the built-in flows, it doesn't preserve user-created flows. As a general best practice, exporting your flows is recommended to create backups of your customizations. Afterwards, you can reimport your flows or reference the exported flow JSON as needed.

2. Stop all running containers:

   Docker

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

   Podman

   ```
   podman stop --all
   ```

3. Remove and clean up containers:

   i. Remove all containers, including stopped containers:

   Docker

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

   Podman

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

ii. Remove all images:

Docker

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

Podman

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

iii. Remove all volumes:

Docker

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

Podman

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

iv. Remove all networks except the default network:

Docker

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

Podman

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

v. Clean up any leftover data:

Docker

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

Podman

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

4. Optional: Edit OpenRAG's `.env` file if needed.

5. Optional: If you removed all containers or specifically the OpenSearch container, then you can remove any files from the `~/.openrag/documents` subdirectory that you don't want to reingest after redeploying the containers. It is recommended that you preserve OpenRAG's default documents.

6. If you removed all OpenRAG containers, redeploy OpenRAG. If you removed only one container, redeploy that container with the appropriate `docker run` or `podman run` command.

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 `uvx` to install OpenRAG, clear your `uv` cache (`uv cache clean`) to remove the TUI environment, and then delete the `~/.openrag` directory.

If you used `uv` to install OpenRAG, run `uv remove openrag` in your Python project, and then delete the `~/.openrag` directory.

Uninstall self-managed deployments

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

Uninstall with `docker compose` or `podman compose`

Use these steps to uninstall a self-managed OpenRAG deployment with streamlined `docker compose` or `podman compose` commands:

1. Destroy the containers, volumes, and local images, and then remove (prune) any additional container 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 OpenRAG's `.env` file and the `~/.openrag/documents` directory, which aren't deleted by the previous commands.

3. Stop `docling-serve`:

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

Uninstall with discrete `docker` or `podman` commands

Use these commands to uninstall a self-managed OpenRAG deployment with discrete `docker` or `podman` commands:

1. Stop all running containers:

Docker

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

Podman

```
podman stop --all
```

2. Remove and clean up containers:

    i. Remove all containers, including stopped containers:

Docker

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

Podman

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

    ii. Remove all images:

Docker

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

Podman

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

iii. Remove all volumes:

Docker

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

Podman

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

iv. Remove all networks except the default network:

Docker

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

Podman

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

v. Clean up any leftover data:

Docker

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

Podman

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

3. Remove OpenRAG's `.env` file and the `~/.openrag/documents` directory, which aren't deleted by the previous commands.

4. 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 `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 then use 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 `uvx`, access the TUI with `uvx openrag`.

You can navigate the TUI with your mouse or keyboard. Keyboard shortcuts for additional menus are printed at the bottom of the TUI screen.

Manage services with the TUI

> **TIP**
> A Docker or Podman VM must be running before you start OpenRAG.
>
> For example, on macOS with Podman, open the Podman Desktop app before you start OpenRAG.

In the TUI, click **Start OpenRAG** to start the OpenRAG services.

Use the TUI's **Status** page to access controls and information for your OpenRAG services.

For more information, see Manage OpenRAG services.

Toggle GPU/CPU mode

You can toggle between GPU and CPU mode from within the TUI if your system has compatible GPU hardware and drivers installed.

In the TUI, click **Status**, and then click **Switch to GPU Mode** or **Switch to CPU Mode**.

This change requires restarting all OpenRAG services because each mode has its own `docker-compose` file.

> **TIP**
> GPU acceleration isn't required for most use cases. OpenRAG's CPU-only deployment doesn't prevent you from using GPU acceleration in external services, such as Ollama servers.
>
> GPU acceleration is required only for specific use cases, typically involving customization of the ingestion flows or ingestion logic. For example, writing alternate ingest logic in OpenRAG that uses GPUs directly in the container, or

customizing the ingestion flows to use Langflow's Docling component with GPU acceleration instead of OpenRAG's Docling Serve service.

Exit the OpenRAG TUI

To exit the OpenRAG TUI, press `q` on the TUI main page.

Exiting the TUI doesn't stop your OpenRAG services. Your OpenRAG services continue to run until they are stopped from within the TUI or by another process that inadvertently stops them.

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 `uv` or `uvx`, then 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 and view logs

TUI-managed services

In the TUI, click **Status** to access diagnostics and controls for all OpenRAG services, including container health, ports, and image versions.

To view streaming logs, click the name of a service, and then press `l`.

For the Docling native service, see Stop, start, and inspect native services.

Self-managed services

For self-managed container services, you can get container logs with `docker compose logs` or `podman logs`.

For the Docling native service, see Stop, start, and inspect native services.

Stop and start containers

TUI-managed services

On the TUI's **Status** page, you can stop, start, and restart OpenRAG's container-based services.

You can also use the **Start OpenRAG** option on the TUI's main menu to start all services at once.

> **TIP**
> A Docker or Podman VM must be running before you start OpenRAG.

How does the TUI start the containers?

When you click **Restart** or **Start Services** in the TUI, the following processes are triggered:

1. OpenRAG automatically detects your container runtime, and then checks if your machine has compatible GPU support by checking for `CUDA`, `NVIDIA_SMI`, and Docker/Podman runtime support. This check determines which Docker Compose file OpenRAG uses because there are separate Docker Compose files for GPU and CPU deployments.
2. OpenRAG pulls the OpenRAG container images with `docker compose pull` if any images are missing.
3. OpenRAG deploys the containers with `docker compose up -d`.

Manually start TUI-managed containers

Alternatively, you can manually run the Docker/Podman commands to stop and start containers. Typically, this is only necessary if there is an error preventing the TUI from managing the containers.

- Stop all containers: `docker compose down` or `podman compose down`
- Start all containers: `docker compose up -d` or `podman compose up -d`
- Stop one container: `docker stop CONTAINER_ID` or `podman stop CONTAINER_ID`
- Start one container: `docker start CONTAINER_ID` or `podman start CONTAINER_ID`

Self-managed services

For self-managed deployments, run the Docker/Podman commands to stop and start containers:

- Stop all containers: `docker compose down` or `podman compose down`
- Start all containers: `docker compose up -d` or `podman compose up -d`
- Stop one container: `docker stop CONTAINER_ID` or `podman stop CONTAINER_ID`
- Start one container: `docker start CONTAINER_ID` or `podman 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-managed services

On the TUI's **Status** page, you can stop, start, restart, and inspect OpenRAG's native services.

The **Native Services** section lists the status, port, and process ID (PID) for each native service.

To manage a native service, click the service's name, and then click **Stop**, **Start** or **Restart**.

To view the logs for a native service, click the service's name, and then press `l`.

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)

> **WARNING**
>
> This is a destructive operation that does the following:
>
> - Destroys all OpenRAG containers, volumes, and local images.
> - Prunes any additional container objects.
> - Deletes the contents of the `~/.openrag` directory *except* for OpenRAG's `.env` file and the `/documents` subdirectory.
>
> Destroyed containers and deleted data are lost and cannot be recovered after running this operation.

Use these steps to reset your OpenRAG deployment by recreating the containers and deleting all data in the `~/.openrag` directory *except* for the `.env` file and the `/documents` subdirectory.

This restores your OpenRAG deployment to a near-initial state while preserving your configuration (in `.env`) and uploaded documents (in `/documents`). Your documents are reingested into a fresh OpenSearch index after the reset.

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

TUI-managed services

1. If you modified the built-in flows or created custom flows in your OpenRAG Langflow instance, export your flows before starting this process. Although OpenRAG can preserve changes to the built-in flows, it doesn't preserve user-created flows. As a general best practice, exporting your flows is recommended to create backups of your customizations. Afterwards, you can reimport your flows or reference the exported flow JSON as needed.

2. To destroy and recreate your OpenRAG containers, click **Status** in the TUI, and then click **Factory Reset**.

3. 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.

4. If you exported customized flows, import your flows into Langflow after completing the onboarding process.

Self-managed services

1. If you modified the built-in flows or created custom flows in your OpenRAG Langflow instance, export your flows before starting this process. Although OpenRAG can preserve changes to the built-in flows, it doesn't preserve user-created flows. As a general best practice, exporting your flows is recommended to create backups of your customizations. Afterwards, you can reimport your flows or reference the exported flow JSON as needed.

2. Recreate the containers:

   Docker

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

Podman

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

3. Launch the OpenRAG app, and then repeat the application onboarding process.

4. If you exported customized flows, import your flows into Langflow after completing the onboarding process.

Prune images

Use image pruning to free up disk space by removing unused OpenRAG container images.

For TUI-managed services, use the TUI's **Prune Images** option to clean up your OpenRAG container images. You can choose to prune unused images only or all images. If you prune all images, the OpenRAG services are stopped, all images are pruned, and then the required images are pulled the next time you start the OpenRAG services.

For self-managed services, use `docker image prune` or `podman image prune` to remove unused images.

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.

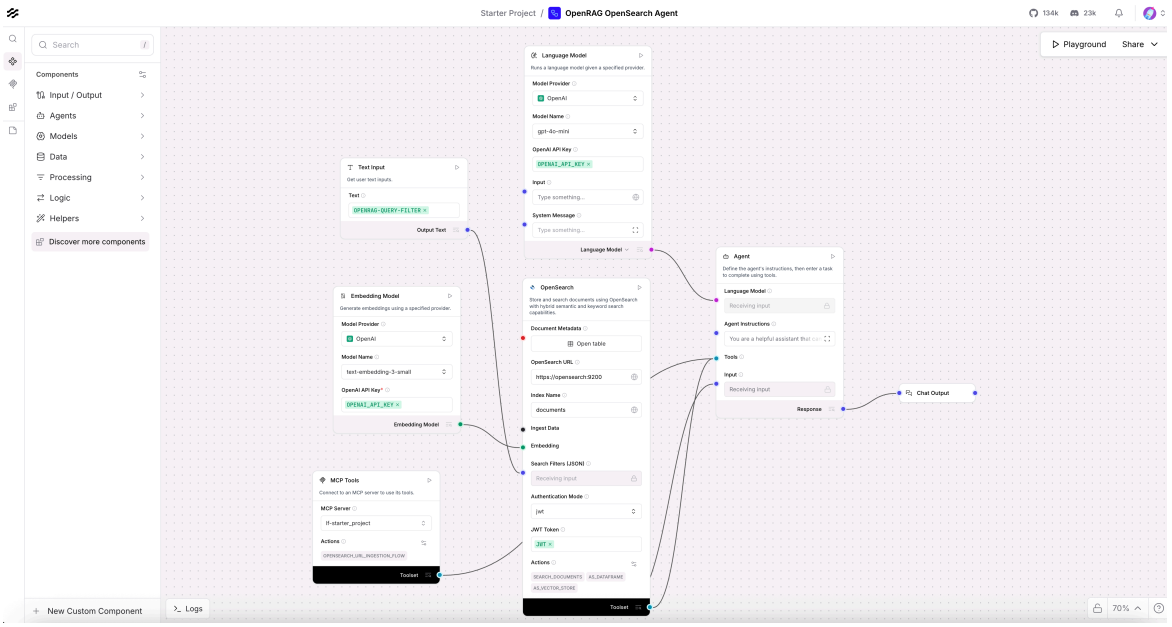
To view and modify a flow in OpenRAG, click **Settings**. From here, you can manage cloud storage connectors, model providers, and common parameters for the **Agent** and **Knowledge Ingestion** flows.

To further explore and edit flows, click **Edit in Langflow** to launch the embedded Langflow visual editor where you can fully customize the flow to suit your use case.

> **TIP**
>
> After you click **Edit in Langflow**, you can access and edit all of OpenRAG's built-in flows from the Langflow editor's **Projects** page. You can also create your own flows.
>
> To edit any flow, you must first unlock the flow.
>
> If you edit any flows other than the **Agent** or **Knowledge Ingestion** flows, it is recommended that you export the flows before editing so you can revert them to their original state if needed.

For example, the following steps explain how to edit the built-in **Agent** flow, which is the **OpenRAG OpenSearch Agent** flow used for the OpenRAG **Chat**:

1. In OpenRAG, click **Settings**, and then find the **Agent** section.
2. If you only need to edit the language model or agent instructions, edit those fields directly on the **Settings** page. Language model changes are saved automatically. To apply new instructions, click **Save Agent Instructions**.
3. To edit all flow settings and components with full customization capabilities, edit the flow in the Langflow visual editor:
    i. Click **Edit in Langflow** to launch the Langflow visual editor in a new browser tab.
    ii. If prompted to acknowledge that you are entering Langflow, click **Proceed**.
    iii. If Langflow requests login information, enter the `LANGFLOW_SUPERUSER` and `LANGFLOW_SUPERUSER_PASSWORD` from your OpenRAG `.env` file.
    iv. In the Langflow header, click **Starter Project** to go to the Langflow **Projects** page, and then unlock the flow.
    v. On the **Projects** page, click the **OpenRAG OpenSearch Agent** flow to open the visual editor with the flow unlocked.



vi. Modify the flow as desired, and then press `Command`+`S` (`Ctrl`+`S`) to save your changes. Then, you can close the Langflow browser tab, or leave it open if you want to continue experimenting with the flow editor.

4. After you modify any **Agent** flow settings, go to the OpenRAG **Chat**, and then click **Start new conversation** in the **Conversations** list. 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 the **Agent** or **Knowledge Ingestion** built-in flows, you can click **Restore flow** on the **Settings** page to revert either flow to its original state when you first installed OpenRAG. This is a destructive action that discards all customizations to the flow.

This option isn't available for other built-in flows such as the **Nudges** flow. To restore these flows to their original state, you must reimport the flow from a backup (if you exported one before editing), or reset or reinstall OpenRAG.

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.

Modify a flow at runtime

You can use *tweaks* to modify flow settings at runtime without permanently changing the flow's configuration. Tweaks 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).

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` environment variable. 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.

About 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 knowledge base 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.

OpenSearch authentication and document access

When you install OpenRAG, you provide the initial configuration values for your OpenRAG services, including authentication credentials for OpenSearch and optional OAuth connectors.

> ⚠ **WARNING**
>
> Google is the only supported OAuth provider for OpenRAG.
>
> Other OAuth credentials are used only to authorize cloud storage connectors.

The presence of Google OAuth credentials determines how OpenRAG authenticates with your deployment's OpenSearch knowledge base, and how it controls user access to documents in your knowledge base:

- **No-auth mode**: If you don't provide Google OAuth credentials, then the OpenRAG OpenSearch knowledge base 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**: If you provide Google OAuth credentials, then the OpenRAG OpenSearch knowledge base 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.

> **TIP**
>
> To enable OAuth mode, you must configure the Google Drive cloud storage connector for document ingestion. This is because OpenRAG uses Google OAuth credentials for both OAuth mode *and* the Google Drive cloud storage connector.
>
> You can enable OAuth mode after initial setup by configuring the Google Drive connector.

OpenSearch index

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`.

To change the OpenRAG index name, edit the `OPENSEARCH_INDEX_NAME` environment variable.

OpenSearch service port

You can access your OpenRAG OpenSearch dashboard directly at `https://localhost:9200`.

To change the port used by the OpenSearch service, edit the `OPENSEARCH_PORT` environment variable.

See also

- Ingest knowledge

- Configure ingestion
- Browse and manage knowledge
- Chat with knowledge
- Troubleshoot document ingestion or similarity search issues

Ingest knowledge

Upload documents to your OpenRAG OpenSearch knowledge base to populate your knowledge base with unique content, such as your own company documents, research papers, or websites.

OpenRAG can ingest knowledge from direct file uploads, URLs, and cloud storage 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 knowledge base.

During ingestion, documents are broken into smaller chunks of content. Embeddings are generated for each chunk so the documents can be retrieved through similarity search during chat, which is a standard retrieval augmented generation (RAG) pattern. The chunks, embeddings, and associated metadata (which connect chunks of the same document) are stored in your OpenSearch knowledge base.

To modify chunking behavior, embedding model, and other ingestion settings, see Configure ingestion.

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 `~/.openrag/documents`. To change this path, see Set the local documents path.

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

About the OpenSearch Ingestion flow

When you upload documents locally or with cloud storage 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 your OpenRAG OpenSearch knowledge base.

The default address for the OpenSearch knowledge base is `https://localhost: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.

Supported file types

When ingesting from the local file system or cloud storage connectors, OpenRAG supports the following file types:

- .adoc
- .asciidoc
- .bmp
- .csv
- .doc
- .docx
- .gif
- .htm
- .html
- .jpeg
- .jpg
- .md
- .odt
- .pdf
- .png
- .ppt
- .pptx
- .rtf
- .tiff
- .txt
- .webp

- .xls
- .xlsx

Ingest local files temporarily

When using the OpenRAG **Chat**, click + **Add** 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 from cloud storage

To ingest knowledge from cloud storage using an OpenRAG cloud storage connector, do the following:

1. Configure a cloud storage connector.
2. Click **Knowledge** to view your OpenSearch knowledge base.
3. Click **Add Knowledge**, and then select a storage provider.
4. On the **Add Cloud Knowledge** page, click **Add Files**, and then select the files and folders to ingest from the connected storage.
5. Click **Ingest Files**.

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

About the OpenSearch Ingestion flow

When you upload documents locally or with cloud storage 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 your OpenRAG OpenSearch knowledge base.

  The default address for the OpenSearch knowledge base is `https://localhost: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

When using the OpenRAG chat, you can enter URLs into the chat to be ingested in real-time during your conversation.

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

> **TIP**
> Like all OpenRAG flows, you can inspect the flow in Langflow, and you can customize it.
>
> To ingest URLs recursively, edit the **Depth** parameter in the **OpenSearch URL Ingestion** flow.

The OpenRAG chat cannot ingest URLs that end in static document file extensions like `.pdf`. To upload these types of files, see Ingest local files and folders and Ingest files with cloud storage connectors.

Monitor ingestion

Depending on the amount of data to ingest, document ingestion can take a few seconds, minutes, or longer. For this reason, 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. Tasks are separated into multiple sections:

- The **Active Tasks** section includes all tasks that are **Pending**, **Running**, or **Processing**:
    - **Pending**: The task is queued and waiting to start.
    - **Running**: The task is actively processing files.

    - **Processing**: The task is performing ingestion operations.

  To stop an active task, click ✕ **Cancel**. Canceling a task stops processing immediately and marks the ingestion as failed.
- The **Recent Tasks** section lists recently finished tasks.

  > ⚠ **WARNING**
  >
  > **Completed** doesn't mean success.
  >
  > A completed task can report successful ingestions, failed ingestions, or both, depending on the number of files processed.

  Check the **Success** and **Failed** counts for each completed task to determine the overall success rate.

  Failed means something went wrong during ingestion, or the task was manually canceled.

For each task, depending on its state, you can find the task ID, start time, duration, number of files processed successfully, number of files that failed, and the number of files enqueued for processing.

Ingestion performance expectations

Ingestion performance depends on many factors, such as the number and size of files ingested, file types, file contents, embedding model, chunk size, and hardware resources.

Particularly when ingesting folders and very large files, such as PDFs with more than 300 pages, ingestion can take a long time or time out. For more information, see Troubleshoot document ingestion or similarity search issues

Example: Ingestion performance test

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 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

See also

- About knowledge
- Configure ingestion
- Configure connectors
- Browse and manage knowledge
- Chat with knowledge
- Troubleshoot document ingestion or similarity search issues

Configure ingestion

The knowledge ingestion settings determine how documents are processed when you upload documents into your knowledge base. This includes chunking strategies, embedding models, and image handling settings.

Existing documents aren't reprocessed after changing ingestion settings

> **WARNING**
> Changes to knowledge ingestion settings only apply to documents that you upload after making 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.

Select a Docling implementation

OpenRAG uses Docling for document ingestion. Docling processes files, splits them into chunks, and stores them as separate, structured documents in your OpenSearch knowledge base.

You can configure OpenRAG to use either a Docling Serve service or a Docling processor pipeline for document processing:

- **Docling Serve ingestion**: By default, OpenRAG uses Docling Serve. When you start OpenRAG, a local `docling serve` process starts, and then OpenRAG runs Docling ingestion through the Docling Serve API.

To use a remote `docling serve` instance or your own local instance, set `DOCLING_SERVE_URL=http://HOST_IP:5001` in your OpenRAG `.env` file. The service must run on port 5001.

For TUI-managed deployments, The TUI warns you if `docling serve` isn't running or isn't detected by OpenRAG. For information about starting and stopping OpenRAG services, see Manage OpenRAG services.

- **Docling processor ingestion**: Instead of using a separate Docling Serve service and the Docling Serve API, you can use the Docling processor directly. To do this, set `DISABLE_INGEST_WITH_LANGFLOW=true` in your OpenRAG `.env` file, and then restart the OpenRAG services. For the underlying functionality for this option, see `processors.py` in the OpenRAG repository.

Set the embedding model and dimensions

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

After onboarding, you can change the embedding model on the OpenRAG **Settings** page. OpenRAG automatically updates all relevant OpenRAG flows to use the new embedding model and dimensions.

You can only select models that are available from your configured model providers, such as an Ollama instance or an OpenAI account. For more information, see Troubleshoot model availability and performance issues.

Because the OpenRAG UI validates model availability and compatibility, setting models directly in the OpenRAG `.env` file isn't recommended.

Best practices for multiple embedding models

For ingestion, OpenRAG allows only one active embedding model at a time.

OpenRAG *doesn't* reprocess existing documents when you change the embedding model. If you want to generate new embeddings for existing documents, you must reupload the documents after enabling the new embedding model. To remove previously

generated embeddings, you must delete the existing document from the knowledge base.

If you use different embedding models for different documents, you can create filters to separate documents that were embedded with different models.

If you use multiple embeddings models, be aware that similarity search (in **Chat**) can take longer as the agent searches each model's embeddings separately.

Set the chunking strategy

You can edit the following settings on the OpenRAG **Settings** page in the **Knowledge Ingest** section:

- **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 with the default **Chunk size** of 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.

Configure table parsing

You can edit the following setting on the OpenRAG **Settings** page in the **Knowledge Ingest** section:

- **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.

Configure OCR and image processing

You can edit the following settings on the OpenRAG **Settings** page in the **Knowledge Ingest** section:

- **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 local documents paths is set when you install OpenRAG and start the OpenRAG services.

The default path for local uploads is `~/.openrag/documents`. 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 either of the following, and then restart the OpenRAG services:

- The **Documents Paths** setting in the **Basic/Advanced Setup** menu
- The `OPENRAG_DOCUMENTS_PATH` variable in the OpenRAG `.env` file.

See also

- About knowledge
- Ingest knowledge
- Browse and manage knowledge

- Configure connectors
- Inspect and modify flows
- Troubleshoot document ingestion or similarity search issues

Browse and manage knowledge

The **Knowledge** page lists the documents that have been ingested into your OpenSearch knowledge base, specifically in an OpenSearch index.

To explore the contents of your knowledge base, click **Knowledge** to get a list of all ingested documents.

Inspect knowledge

For each document, the **Knowledge** page provides the following information:

- **Source**: Name of the ingested content, such as the file name.
- **Size**
- **Type**
- **Owner**: User that uploaded the document.

  In no-auth mode, all documents are attributed to **Anonymous User** because there is no distinct document ownership or unique JWTs. For more control over document ownership and visibility, use OAuth mode. For more information, see OpenSearch authentication and document access.
- **Chunks**: Number of chunks created by splitting the document during ingestion.

  Click a document to view the individual chunks and technical details related to chunking. If the chunks seem incorrect or incomplete, see Troubleshoot document ingestion or similarity search issues.
- **Avg score**: Average similarity score across all chunks of the document.

  If you search the knowledge base, the **Avg score** column shows the similarity score for your search query or filter.
- **Embedding model** and **Dimensions**: The embedding model and dimensions used to embed the chunks. The embedding model, dimensions, chunks, and other ingestion

behaviors are determined by the knowledge ingestion settings that you set before uploading a document.
- **Status**: Status of document ingestion. If ingestion is complete and successful, then the status is **Active**. For more information, see Monitor ingestion.

Search knowledge

You can use the search field on the **Knowledge** page to find documents using semantic search and knowledge filters:

To search all documents, enter a search string in the search field, and then press `Enter`.

To apply a knowledge filter, select the filter from the **Knowledge Filters** list. The filter settings pane opens, and the filter appears in the search field. To remove the filter, close the filter settings pane or clear the filter from the search field.

You can use the filter alone or in combination with a search string. If a knowledge filter has a **Search Query**, that query is applied in addition to any text string you enter in the search field.

Only one filter can be applied at a time.

Default documents

By default, OpenRAG includes some initial documents about OpenRAG. These documents are ingested automatically during the application onboarding process if you use the default local documents path.

You can use these documents to ask OpenRAG about itself, or to test the **Chat** feature before uploading your own documents.

If you delete these documents, then you won't be able to ask OpenRAG about itself and its own functionality unless you re-ingest those documents. It is recommended that you keep these documents, and use filters to separate them from your other knowledge. An **OpenRAG Docs** filter is created automatically for these documents.

Add knowledge

Add knowledge to your knowledge base by ingesting documents.

Delete knowledge

> **WARNING**
> This is a destructive operation that cannot be undone.

To delete documents from your knowledge base, click **Knowledge**, use the checkboxes to select one or more documents, and then click **Delete**. If you select the checkbox at the top of the list, all documents are selected and your entire knowledge base will be deleted.

To delete an individual document, you can also click **More** next to that document, and then select **Delete**.

To completely clear your entire knowledge base and OpenSearch index, reset your OpenRAG containers or reinstall OpenRAG.

See also

- About knowledge
- Ingest knowledge
- Filter knowledge
- Troubleshoot document ingestion or similarity search issues

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**.
3. Optional: Click the filter icon next to the filter name to select a different icon and color for the filter. This is purely cosmetic, but it can help you visually distinguish different sets of filters, such as different projects or sources.
4. Optional: Enter a **Description**.

5. Customize the filter settings.

   By default, filters match all documents in your knowledge base. Use the filter settings to narrow the scope of documents that the filter captures:

   - **Search Query**: Enter a natural language text string for semantic search.

     When you apply a filter that has a **Search Query**, only documents matching the search query are included. It is recommended that you also use the **Score Threshold** setting to avoid returning irrelevant documents.

   - **Data Sources**: Select specific files and folders to include in the filter.

     This is useful if you want to create a filter for a specific project or topic and you know the specific documents you want to include. For example, if you upload a folder, you could create a filter that only includes the documents from that folder.

   - **Document Types**: Filter by file type.

   - **Owners**: Filter by the user that uploaded the documents.

     In no-auth mode, all documents are attributed to **Anonymous User** because there is no distinct document ownership or unique JWTs. For more control over document ownership and visibility, use OAuth mode. For more information, see OpenSearch authentication and document access.

   - **Connectors**: Filter by upload source, such as the local file system or a cloud storage connector.

   - **Response Limit**: Set the maximum number of results to return from the knowledge base. The default is `10`, which means the filter returns only the top 10 most relevant documents.

   - **Score Threshold**: Set the minimum relevance score for similarity search. The default score is `0`. A threshold is recommended to avoid returned irrelevant documents.

6. Click **Create Filter**.

Edit a filter

To modify a filter, click **Knowledge**, and then click the filter you want to edit in the **Knowledge Filters** list. On the filter settings pane, edit the filter as desired, and then click **Update Filter**.

Apply a filter

In the OpenRAG **Chat**, click **Filter**, and then select the filter to apply. Chat filters apply to one chat session only.

You can also use filters when browsing your knowledge base. This is a helpful way to test filters and manage knowledge bases that have many documents.

Delete a filter

1. Click **Knowledge**.
2. In the **Knowledge Filters** list, click the filter that you want to delete.
3. In the filter settings pane, click **Delete Filter**.

See also

- About knowledge
- Browse and manage knowledge
- Configure connectors
- Chat with knowledge

Configure connectors

OpenRAG uses OAuth credentials to authorize access to cloud storage services so you can ingest documents from cloud storage.

If you provide Google OAuth credentials, these credentials are also used to enable OAuth mode for OpenRAG and your OpenSearch knowledge base.

OAuth credentials and OpenSearch authentication modes

When you install OpenRAG, you provide the initial configuration values for your OpenRAG services, including authentication credentials for OpenSearch and optional OAuth connectors.

> **WARNING**
>
> Google is the only supported OAuth provider for OpenRAG.
>
> Other OAuth credentials are used only to authorize cloud storage connectors.

The presence of Google OAuth credentials determines how OpenRAG authenticates with your deployment's OpenSearch knowledge base, and how it controls user access to documents in your knowledge base:

- **No-auth mode**: If you don't provide Google OAuth credentials, then the OpenRAG OpenSearch knowledge base 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**: If you provide Google OAuth credentials, then the OpenRAG OpenSearch knowledge base 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.

> **TIP**

> To enable OAuth mode, you must configure the Google Drive cloud storage connector for document ingestion. This is because OpenRAG uses Google OAuth credentials for both OAuth mode *and* the Google Drive cloud storage connector.
>
> You can enable OAuth mode after initial setup by configuring the Google Drive connector.

Cloud storage connectors

You can use OpenRAG's cloud storage connectors to ingest files from cloud storage. Specifically, you can connect to the following services:

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

To configure a cloud storage connector, you must register an OAuth app, get OAuth credentials, and then add the OAuth credentials to your OpenRAG configuration.

Register an OAuth app and generate credentials

Register OpenRAG as an OAuth application in your cloud provider. Then, obtain OAuth credentials for the app, such as a client ID and secret key.

To enable multiple connectors, you must register an app and generate credentials for each provider.

Add OAuth credentials to OpenRAG

TUI-managed services

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

1. If OpenRAG is running, click **Stop All Services** in the TUI.

2. Open the **Advanced Setup** page, and then add the OAuth credentials for the cloud storage providers that you want to use under **API Keys**:

- **Google**: Enter 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.

  Providing these Google credentials enables OAuth mode *and* the Google Drive cloud storage connector.

  > **WARNING**
  >
  > Google is the only supported OAuth provider for OpenRAG.
  >
  > You must enter Google credentials if you want to enable OAuth mode.
  >
  > The Microsoft and Amazon credentials are used only to authorize the cloud storage connectors. OpenRAG doesn't offer OAuth provider integrations for Microsoft or Amazon.

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

- **Amazon**: Enter 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.

3. For each connector you configured, register the redirect URIs shown in the TUI in your OAuth apps.

   The redirect URIs are used for the cloud storage connector webhooks. For Google, the redirect URIs are also used to redirect users back to OpenRAG after they sign in.

4. Optional: Under **Others**, set the **Webhook Base URL** to the base address for your OAuth connector endpoints. If set, the OAuth connector webhook URLs are constructed as `WEBHOOK_BASE_URL/connectors/${provider}/webhook`. This option is required to enable automatic ingestion from cloud storage.

5. Click **Save Configuration** to add the OAuth credentials to your OpenRAG `.env` file.

6. Click **Start Services** to restart the OpenRAG containers with the new configuration.

7. Launch the OpenRAG app.

    If you provided Google OAuth credentials, you must sign in with Google before you are redirected to your OpenRAG instance.

Self-managed services

If you installed OpenRAG with self-managed services, set OAuth credentials in your OpenRAG `.env` file.

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 your OpenRAG `.env` file, and then add the OAuth and cloud storage environment variables for the providers that you want to use:

    ```
    GOOGLE_OAUTH_CLIENT_ID=
    GOOGLE_OAUTH_CLIENT_SECRET=
    MICROSOFT_GRAPH_OAUTH_CLIENT_ID=
    MICROSOFT_GRAPH_OAUTH_CLIENT_SECRET=
    AWS_ACCESS_KEY_ID=
    AWS_SECRET_ACCESS_KEY=
    ```

    - **Google**: Enter 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.

Providing these Google credentials enables OAuth mode *and* the Google Drive cloud storage connector.

> **WARNING**
>
> Google is the only supported OAuth provider for OpenRAG.
>
> You must enter Google credentials if you want to enable OAuth mode.
>
> The Microsoft and Amazon credentials are used only to authorize the cloud storage connectors. OpenRAG doesn't offer OAuth provider integrations for Microsoft or Amazon.

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

- **Amazon**: Enter 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.

3. Optional: Set the `WEBHOOK_BASE_URL` to the base address for your OAuth connector endpoints. If set, the OAuth connector webhook URLs are constructed as `WEBHOOK_BASE_URL/connectors/${provider}/webhook`. This option is required to enable automatic ingestion from cloud storage.

4. Save the `.env` file.

5. For each connector, you must register the OpenRAG redirect URIs in your OAuth apps:

   - Local deployments: `http://localhost:3000/auth/callback`
   - Production deployments: `https://your-domain.com/auth/callback`

   The redirect URIs are used for the cloud storage connector webhooks. For Google, the redirect URIs are also used to redirect users back to OpenRAG after they sign in.

6. Restart your OpenRAG containers:

Docker

```
docker compose up -d
```

Podman

```
podman compose up -d
```

7. Access the OpenRAG frontend at `http://localhost:3000`.

   If you provided Google OAuth credentials, you must sign in with Google before you are redirected to your OpenRAG instance.

Ingest documents with cloud connectors

See Ingest files from cloud storage.

Disconnect and reconnect cloud storage connectors

In OpenRAG, click **Settings** to manage cloud storage connections. You can connect, disconnect, reconnect, and trigger ingestion for each configured connector.

See also

- About knowledge
- Ingest knowledge
- Configure ingestion
- Browse and manage knowledge

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.

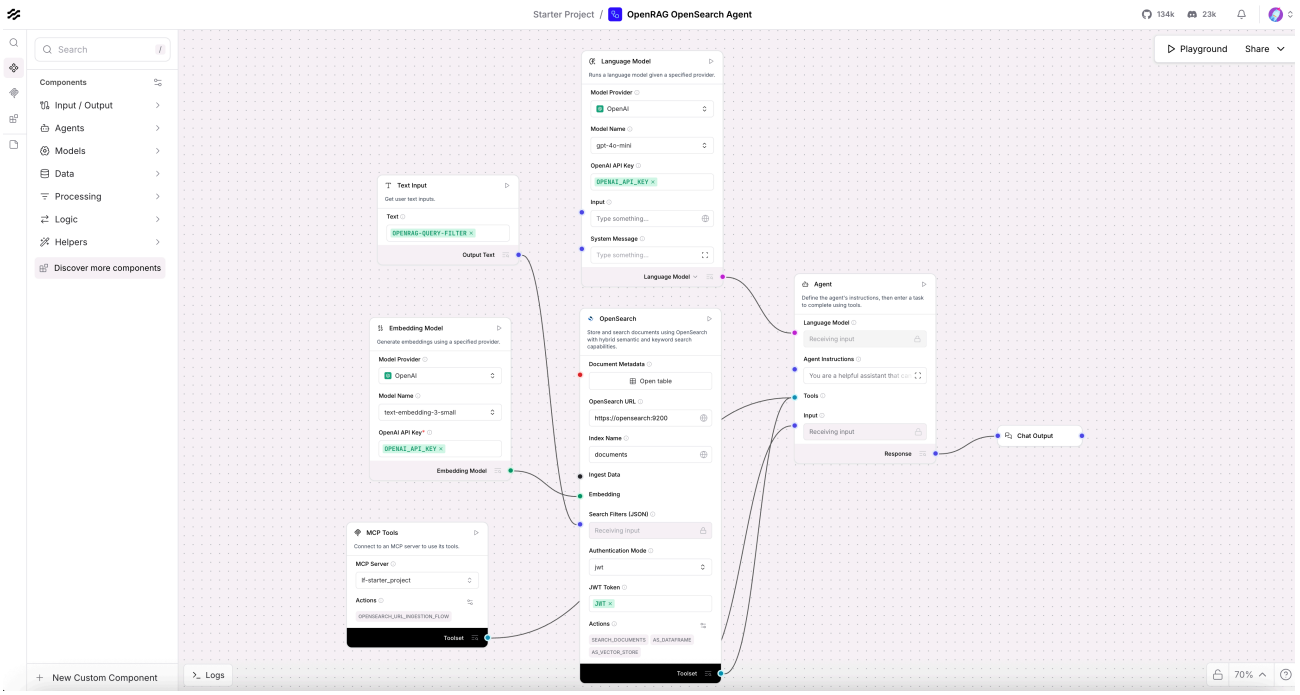
The agent can call specialized Model Context Protocol (MCP) tools to extend its capabilities. To add or change the available tools, you must edit the **OpenRAG OpenSearch Agent** flow.

> **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.

  > **HOW DO AGENTS WORK?**
  >
  > Agents extend Large Language Models (LLMs) by integrating tools, which are functions that provide additional context and enable autonomous task execution. These integrations make agents more specialized and powerful than standalone LLMs.
  >
  > Whereas an LLM might generate acceptable, inert responses to general queries and tasks, an agent can leverage the integrated context and tools to provide

> more relevant responses and even take action. For example, you might create an agent that can access your company's documentation, repositories, and other resources to help your team with tasks that require knowledge of your specific products, customers, and code.
>
> Agents use LLMs as a reasoning engine to process input, determine which actions to take to address the query, and then generate a response. The response could be a typical text-based LLM response, or it could involve an action, like editing a file, running a script, or calling an external API.
>
> In an agentic context, tools are functions that the agent can run to perform tasks or access external resources. A function is wrapped as a Tool object with a common interface that the agent understands. Agents become aware of tools through tool registration, which is when the agent is provided a list of available tools typically at agent initialization. The Tool object's description tells the agent what the tool can do so that it can decide whether the tool is appropriate for a given request.

- **Language Model** component: Connected to the **Agent** component's **Language Model** port, this component provides the base language model driver for the agent. The agent cannot function without a model because the model is used for general knowledge, reasoning, and generating responses.

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

- **MCP Tools** component: Connected to the **Agent** component's **Tools** port, this component can be used to access any 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, and they are based on the contents of your OpenRAG OpenSearch knowledge base. Click a nudge to accept it and start a chat based on the nudge.

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. However, this flow is specifically

designed to work with the OpenRAG chat and knowledge base. Major changes to this flow might break the nudge functionality or produce irrelevant nudges.

The **Nudges** flow consists of **Embedding model**, **Language model**, **OpenSearch**, **Input/Output*, and other components that browse your knowledge base, identify key themes and possible insights, and then produce prompts based on the findings.

For example, if your knowledge base contains documents about cybersecurity, possible nudges might include `Explain zero trust architecture principles` or `How to identify a social engineering attack`.

Upload documents to the chat

When using the OpenRAG **Chat**, click + **Add** 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 document ingestion or similarity search issues.

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 OpenRAG SDKs (recommended) or the Langflow API.

> **INFO**
>
> It is strongly recommended that you use the OpenRAG SDKs to interact with OpenRAG programmatically or integrate OpenRAG into an application. This is because OpenRAG passes specific configuration settings to the Langflow API when it runs a flow or conducts a chat session. For more information, see the following:
>
> - Python SDK README
> - TypeScript/JavaScript SDK README

Use the Langflow API (Not recommended)

To trigger a flow with the Langflow API, you can get pre-configured code snippets directly from the embedded Langflow visual editor. However, these snippets don't include OpenRAG-specific configuration settings, so the behavior won't be identical to using OpenRAG directly.

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:
    i. In OpenRAG, click **Settings**.
    ii. Click **Edit in Langflow**, and then click **Proceed** to launch the Langflow visual editor.
    iii. If Langflow requests login information, enter the `LANGFLOW_SUPERUSER` and `LANGFLOW_SUPERUSER_PASSWORD` from your OpenRAG `.env` file.
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.

See also

- Troubleshoot chat issues
- Troubleshoot document ingestion or similarity search issues

Environment variables

OpenRAG's `.env` file is the primary configuration file for OpenRAG. Environment variables in `.env` always take precedence over other sources.

For deployments managed with the Terminal User Interface (TUI), this file is located at `~/.openrag/tui`, and it can be created automatically during installation.

For self-managed deployments, this file can be located at the root of your OpenRAG project directory or referenced from another location.

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

> **TIP**
> OpenRAG's Docker Compose files are populated automatically using values from the `.env` file, so you don't need to edit the Docker Compose files manually.

If a variable isn't set, OpenRAG uses default or fallback values where available. Not all variables have default values, and errors can occur if required variables aren't set. Default values can be found in the reference tables on this page and in `config_manager.py`, `settings.py`, and `logging_config.py`.

You can temporarily set Langflow variables at runtime. However, these temporary overrides don't overlap with most OpenRAG environment variables. The only exceptions are flow-level Langflow settings, such as the language model used in a flow.

Edit the `.env` file and apply configuration changes

During installation, you prepare an initial `.env` file. For TUI-managed deployments, this file is created automatically. For self-managed deployments, you create this file manually.

After installation, you can edit the `.env` file to change your OpenRAG configuration.

> **TIP**

> For TUI-managed deployments, many environment variables can be set in the TUI's When you save the changes and restart the OpenRAG services, the `.env` is updated automatically.

Most OpenRAG environment variables are mutable, and you can apply changes to these settings by stopping and restarting the OpenRAG services after editing the `.env` file.

If a change doesn't take effect after restarting the services, then the variable is immutable, and you must redeploy OpenRAG with your modified `.env` file. This is typically only required for critical configuration changes that affect the core OpenRAG infrastructure.

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 the application onboarding process.

Variable	Default	Description
`EMBEDDING_PROVIDER`	`openai`	Embedding model provider, as one of `openai`, `watsonx`, or `ollama`.
`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 the application onboarding process.
`LLM_PROVIDER`	`openai`	Language model provider, as one of `openai`, `watsonx`, `ollama`, or `anthropic`.
`LLM_MODEL`	`gpt-4o-mini`	Language model for language processing and text generation in the **Chat** feature. Can be changed after the application onboarding process.
`ANTHROPIC_API_KEY`	Not set	API key for the Anthropic model provider.

Variable	Default	Description
`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

Controls some aspects of how OpenRAG processes and ingests documents into your knowledge base.

For the embedding model and Docling engine variables that you can set on the OpenRAG **Settings** page, see Configure ingestion.

For Langflow flow IDs and Langflow timeout settings, see Langflow settings.

Variable	Default	Description
`DISABLE_INGEST_WITH_LANGFLOW`	`false`	Disable Langflow ingestion pipeline i you don't want to u the default Docling Serve instance.
`DOCLING_SERVE_URL`	`http://HOST_IP:5001`	URL for the Doclin Serve instance. By default, OpenRAG starts a local `docling serve` process and auto-detects the host. T use your own local remote Docling Serve instance, se

Variable	Default	Description
		this variable to the full path to the targ instance. The serv must run on port 5001.
`OPENRAG_DOCUMENTS_PATH`	`~/.openrag/documents`	The local documer path for ingestion.
`HOST_DOCKER_INTERNAL`	`host.docker.internal`	Host address for accessing services running on the hos machine from with Docker containers This is used to connect to the loc Docling Serve instance started b OpenRAG. If your system uses a different hostname for this purpose, s this variable accordingly.
`INGESTION_TIMEOUT`	`3600`	Document ingestic timeout limit in seconds for each f Increase this value you experience timeouts when ingesting very larg documents. Must l greater than or equ to `LANGFLOW_TIMEO`
`UPLOAD_BATCH_SIZE`	`25`	When ingesting folders, set the maximum number files to ingest per batch. Each batch an ingestion task. Increase this value ingest more files p batch. If this value too high, performance issue can occur.

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 the Lang `LANGFL` set, ther `True` ar `LANGFL` then `LA` and autc Langflov authenti regardle
`LANGFLOW_ENABLE_SUPERUSER_CLI`	Determined by `LANGFLOW_SUPERUSER_PASSWORD`	Whether `langfl` `LANGFL` set, ther `LANGFL` `True` ar created `LANGFL` then `LANGFL` `False` a commar

Variable	Default	
`LANGFLOW_NEW_USER_IS_ACTIVE`	Determined by `LANGFLOW_SUPERUSER_PASSWORD`	Whether active by `LANGFL` set, ther `LANGFL` `True` ar by defau `LANGFL` then `LA` is `False` inactive
`LANGFLOW_PORT`	`7860`	Host por Change on your must als the new
`LANGFLOW_PUBLIC_URL`	`http://localhost:7860`	Public U Forms th calls and OpenRA set in th
`LANGFLOW_SECRET_KEY`	Automatically generated	Secret e internal generate for this v then Lar automat
`LANGFLOW_SUPERUSER`	`admin`	Usernam user.
`LANGFLOW_SUPERUSER_PASSWORD`	Not set	Langflov variable server st enabled. `LANGFL` Langflow authenti
`LANGFLOW_CHAT_FLOW_ID`, `LANGFLOW_INGEST_FLOW_ID`, `NUDGES_FLOW_ID`, `LANGFLOW_URL_INGEST_FLOW_ID`	Built-in flow IDs	These va the IDs c URL inge default v `.env.ex`

Variable	Default	
		values if flow with JSON m the Oper you depl can add of the O deployin
`LANGFUSE_SECRET_KEY`	Not set	Optional the Lang
`LANGFUSE_PUBLIC_KEY`	Not set	Optional the Lang
`LANGFUSE_HOST`	Not set	Leave er Required deploym and `LAN` address containe `http:/` `http:/`
`LANGFLOW_TIMEOUT`	`2400`	Total La seconds experien large PD `INGEST`
`LANGFLOW_CONNECT_TIMEOUT`	`30`	Langflov in secon experien large PD

OAuth and cloud storage connector settings

Use these variables to enable OAuth mode and configure cloud storage connectors.

> **WARNING**
>
> Additional configuration is required for these features. For more information, see Configure connectors.

Variable	Default	Description
`AWS_ACCESS_KEY_ID` `AWS_SECRET_ACCESS_KEY`	Not set	Authorize OpenRAG to ingest documents from AWS S3 with an AWS OAuth app integration.
`GOOGLE_OAUTH_CLIENT_ID` `GOOGLE_OAUTH_CLIENT_SECRET`	Not set	Enable OAuth mode *and* the Google Drive cloud storage connector with 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	Optional base URL for OAuth connector webhook endpoints. If not set, a default base URL is used. This variable is required to enable automatic cloud storage ingestion. Can also be set in the TUI's **Advanced Setup**.

OpenSearch settings

Configure OpenSearch database authentication.

Variable	Default	Description
`OPENSEARCH_DATA_PATH`	`./opensearch-data`	The path where OpenRAG creates your OpenSearch index data. This persists through updates.
`OPENSEARCH_PASSWORD`	Not set	Required. OpenSearch administrator password. Must adhere to the OpenSearch

Variable	Default	Description
		password complexity requirements. You must set this directly in the `.env` or in the TUI's **Basic/Advanced Setup**.
`OPENSEARCH_INDEX_NAME`	`documents`	The name of the OpenSearch index.

System settings

Configure general system components, session management, and logging.

Variable	Default	Description
`FRONTEND_PORT`	`3000`	Host port for the OpenRAG frontend web interface. Change this if port 3000 is already in use on your system.
`OPENRAG_VERSION`	`latest`	The version of the OpenRAG Docker images to run. For more information, see Upgrade OpenRAG
`NEXT_ALLOWED_DEV_ORIGINS`	`http://localhost:3000`	Only used when running OpenRAG in development mode. Accepts a comma-separated list of hostnames to allow additional origins to make requests to the Next.js development server.

Variable	Default	Description
`MAX_WORKERS`	`min(4, CPU_COUNT // 2)`	Number of Backend worker processes for concurrent request handling. Be mindful of hardware limitations to avoid overtaxing system resources.
`LANGFLOW_WORKERS`	`1`	Number of Langflow worker processes for concurrent request handling. Be mindful of hardware limitations to avoid overtaxing system resources.
`DOCLING_WORKERS`	`1`	Number of Docling worker processes for concurrent request handling. Be mindful of hardware limitations to avoid overtaxing system resources.
`ACCESS_LOG`	`true`	Whether to enable access logging for OpenRAG services, such as `INFO: 127.0.0.1:45132 - "GET /tasks HTTP/1.1" 200 OK`. Access logs provide information about incoming requests, and they can be useful for monitoring and debugging.

OpenRAG SDKs

Use the OpenRAG SDKs to interact with OpenRAG programmatically or integrate OpenRAG into an application:

- OpenRAG MCP Server
- Python SDK README
- TypeScript/JavaScript SDK README

For the core APIs, see `main.py`

Contribute to OpenRAG

There are several ways you can interact with the OpenRAG community and contribute to the OpenRAG codebase.

Star OpenRAG on GitHub

If you like OpenRAG, you can star the OpenRAG GitHub repository. Stars help other users find OpenRAG more easily, and quickly understand that other users have found it useful.

Because OpenRAG is open-source, the more visible the repository is, the more likely the codebase is to attract contributors.

Watch the GitHub repository

You can watch the OpenRAG GitHub repository to get notified about new releases and other repository activity.

To get release notifications only, select **Releases only**.

If you select **Watching**, you will receive notifications about new releases as well as issues, discussions, and pull requests. For information about customizing repository notifications, see the GitHub documentation on repository subscriptions.

Request enhancements and get help through GitHub

You can report bugs, submit feature requests, and get help with OpenRAG through the GitHub repository.

The repository is the best place to report bugs and request enhancements to ensure that they are tracked by OpenRAG maintainers.

GitHub issues

The Issues page in the OpenRAG repository is actively updated with bugs and feature requests.

> **TIP**

> The best way to promote a request or bug is to comment on an existing issue that matches your request.
>
> Before you report a bug or submit a feature request, search for existing similar issues. If you find one, add a comment with any relevant details instead of opening a new issue.
>
> Highly active issues are more likely to receive attention from contributors.

Feature planning for OpenRAG is tracked through the Projects page in the OpenRAG repository.

GitHub discussions

If you need help with your code or OpenRAG in general, you can visit the OpenRAG GitHub Discussions page.

The OpenRAG team doesn't provide individual support over email, and the team believes that public discussions help more users by virtue of their discoverability.

Community guidelines and tips

Because the OpenRAG repository is public, the OpenRAG team asks that you follow these guidelines when submitting questions and issues:

- **Provide as many details as possible**: Simply stating that a feature doesn't work isn't helpful. The OpenRAG team needs details in order to recreate and find the issue.
- **Explain what exactly went wrong**: Include error messages and descriptions of *how* your code failed, not just the fact that it failed.
- **Retrace your steps**: Explain what happened before the error, what you expected to happen instead of the error, and any recent changes you made, such as upgrading OpenRAG or a dependency.
- **Describe your environment**: Include your operating system, OpenRAG version, Python version, and any other environment-related details that could have contributed to the issue.
- **Include snippets of the code that failed**: Be sure to omit any sensitive values, and only provide parts that are relevant to the failure rather than the entire script.

Providing code snippets makes it much easier to reproduce errors, troubleshoot, and provide specific advice.
    - If your submission includes long sections of code, logs, or tracebacks, wrap them in details tags (`<details> PASTE CODE HERE </details>`) to collapse the content and make it easier to read your submission.
- **Omit sensitive information**: Other than the information available on your public GitHub profile, don't include sensitive or personally identifying data, such as security keys, full names, personal identification numbers, addresses, and phone numbers.
- **Be kind**: Although bugs can be frustrating with any software, remember that your messages are read by real people who want to help. While you don't have to be saccharine, there's no need to be rude to get support.
    - Your issues and discussions are attached to your GitHub account, and they can be read by anyone on the internet, including current and potential employers and colleagues.
    - The OpenRAG repository is a public GitHub repository and, therefore, subject to the GitHub Code of Conduct.

Contribute to the codebase

If you want to contribute code to OpenRAG, you can do so by submitting a pull request (PR) to the OpenRAG GitHub repository.

See CONTRIBUTING.md to set up your development environment and learn about the contribution process.

Tips for successful submissions

- Explain the motivation for your submission in the PR description. Clarify how the change benefits the OpenRAG codebase and its users.
- Keep PRs small and focused on a single change or set of related changes.
- If applicable, include tests that verify your changes.
- Add documentation for new features, or edit existing documentation (if needed) when modifying existing code. For more information, see Contribute documentation.

Use of AI tools in contributions

If you use AI tools to generate significant portions of the code in your PR, the OpenRAG team asks that you do the following:

- Consider disclosing significant use of AI tools in your pull request description, particularly if you are unable to expertly review your own code contributions. For example: `I used an AI tool to generate the code for <function>, but I am not an expert in <language>. There might be some inefficiencies or antipatterns present.`
- Avoid using AI tools to generate large volumes of code if you don't have personal knowledge of that language and the functionality being implemented. Instead, consider submitting a feature request to the Issues page in the OpenRAG repository.
- Be critical when reviewing code or documentation generated by AI tools to ensure it is accurate, efficient, and avoids antipatterns and vulnerabilities.
- Don't flood the repository with AI-generated pull requests. Low quality and spam contributions can be closed without review at the discretion of the maintainers. Repeated low-quality contributions can lead to a ban on contributions.

Contribute documentation

The OpenRAG documentation is built using Docusaurus and written in Markdown. For style guidance, see the Google Developer Documentation Style Guide.

1. Install Node.js.

2. Fork the OpenRAG GitHub repository.

3. Add the new remote to your local repository on your local machine:

   ```
   git remote add FORK_NAME
   https://github.com/GIT_USERNAME/openrag.git
   ```

   Replace the following:

   - `FORK_NAME`: A name for your fork of the repository
   - `GIT_USERNAME`: Your Git username

4. Change to the `/docs` directory in your local repository:

   ```
   
   ```

```
cd openrag/docs
```

If you're running a development container for code contributions, run the documentation build from outside the container on your host terminal. The documentation build might not work properly when run from inside the development container workspace.

5. Install dependencies and start a local Docusaurus static site with hot reload:

```
npm install
npm start
```

The documentation is served at `http://localhost:3000`.

6. To edit and create content, work with the `.mdx` files in the `openrag/docs/docs` directory.

Create new files in `.mdx` format.

Navigation is defined in `openrag/docs/sidebars.js`.

Most pages use a `slug` for shorthand cross-referencing, rather than supplying the full or relative directory path. For example, if a page has a `slug` of `/cool-page`, you can link to it with `[Cool page](/cool-page)` from any other `/docs` page.

7. Recommended: After making some changes, run `npm run build` to build the site locally with more robust logging. This can help you find broken links before creating a PR.

8. Create a pull request against the `main` branch of the OpenRAG repository with a clear title and description of your changes:

    - Provide a clear title in the format of `Docs: <summary of change>`. For example, `Docs: fix broken link on contributing page`. Pull request titles appear in OpenRAG's release notes, so they should explain what the PR does as explicitly as possible.
    - Explain why and how you made the changes in the pull request description.

    - If the pull request closes an issue, include `Closes #NUMBER` in the description, such as `Closes #1234`.
    - If you used AI tools to write significant portions of the documentation, consider disclosing this in the PR description.
9. Add the `documentation` label to your pull request.
10. Keep an eye on your pull request in case an OpenRAG maintainer requests changes or asks questions.

OpenRAG technical writers can directly edit documentation PRs to enforce style guidelines and fix errors.

Troubleshoot OpenRAG

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

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 `3.12`.

No container runtime found

When you start the TUI, a `No container runtime found` error indicates that OpenRAG cannot find a running Docker or Podman machine.

Make sure Docker or Podman is installed, available in the PATH, and a VM is running.

Container out of memory errors

If you encounter container memory errors, try the following:

- Increase your Podman or Docker VM's allocated memory.

  If you're using Podman on macOS, you might need to increase VM memory on your Podman machine. This example, the following commands stop Podman, increase the machine size to 8 GB of RAM (the minimum recommended RAM for OpenRAG), and then restart the VM:

  ```
  podman machine stop
  podman machine rm
  ```

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

You must also restart your OpenRAG services after increasing the container VM's memory.

- Use a CPU-only deployment to reduce memory usage.

  For TUI-managed deployments, you can enable **CPU mode** on the TUI's **Status** page.

  For self-managed deployments, CPU-only deployments use the `docker-compose.yml` file that doesn't have GPU overrides.

OpenSearch fails to start

Check that the value of the `OPENSEARCH_PASSWORD` environment variable meets the OpenSearch password complexity requirements.

If you need to change the password, you must reset the OpenRAG services.

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.

Application onboarding gets stuck or fails

If the application onboarding process hangs for a long time or fails for an unspecified reason, try the following:

- Make sure you have enough free space (more than 50 GB) available for the temporary storage required for document ingestion.

- Clear your browser's cache.

Langflow connection issues

Verify that the value of the `LANGFLOW_SUPERUSER` environment variable is correct. For more information about this variable and how this variable controls Langflow access, see Langflow settings.

Port conflicts

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

- `3000`: OpenRAG frontend
- `7860`: Langflow service
- `8000`: OpenRAG backend
- `9200`: OpenSearch service
- `5601`: OpenSearch dashboards
- `5001`: Docling service

If the default ports for the OpenRAG frontend (`3000`) or Langflow (`7860`) are already in use on your host machine, you can set the `FRONTEND_PORT` or `LANGFLOW_PORT` environment variables to map these services to different host ports. If you set `LANGFLOW_PORT`, you must also set `LANGFLOW_PUBLIC_URL` to use the new port. To apply the port mapping, restart the containers after editing your OpenRAG `.env` file.

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.

Model availability and performance issues

The following issues relate to the models you can use with OpenRAG and potential performance issues, such as malformed responses and excessive hallucinations.

See also Chat issues.

Language model isn't listed in OpenRAG settings or application onboarding

If your language model isn't listed in the OpenRAG settings or application onboarding, then the model likely doesn't support tool calling, which is required for OpenRAG. You

must select a different model. If no other models are listed, make sure your model provider API key or instance has access to models that support tool calling.

You can submit an OpenRAG GitHub issue to request support for specific models.

IBM watsonx.ai model issues

OpenRAG isn't guaranteed to be compatible with all models that are available through IBM watsonx.ai.

Language models must support tool calling to be compatible with OpenRAG. Incompatible models aren't listed in OpenRAG's settings or onboarding.

Additionally, models must be able to handle the agentic reasoning tasks required by OpenRAG. Models that are too small or not designed for agentic RAG tasks can return low quality, incorrect, or improperly formatted responses. For more information, see Chat issues.

You can submit an OpenRAG GitHub issue to request support for specific models.

Ollama model issues

OpenRAG isn't guaranteed to be compatible with all models that are available through Ollama. Some models might produce unexpected results, such as JSON-formatted output instead of natural language responses, and some models aren't appropriate for the types of tasks that OpenRAG performs, such as those that generate media.

- **Language models**: Ollama-hosted language models must support tool calling to be compatible with OpenRAG. The OpenRAG team recommends `gpt-oss:20b` or `mistral-nemo:12b`. If you choose `gpt-oss:20b`, consider using Ollama Cloud or running Ollama on a remote machine because this model requires at least 16GB of RAM.
- **Embedding models**: The OpenRAG team recommends `nomic-embed-text:latest`, `mxbai-embed-large:latest`, or `embeddinggemma:latest`.

You can experiment with other models, but if you encounter issues that you are unable to resolve through other RAG best practices (like context filters and prompt engineering),

try switching to one of the recommended models. You can submit an OpenRAG GitHub issue to request support for specific models.

Document ingestion or similarity search issues

The following issues can occur during document ingestion or as a result of suboptimal ingestion.

Failed or slow ingestion

If an ingestion task fails, try the following:

- Make sure you ingest only supported file types.
- Split very large files into smaller files.
- Remove unusual or complex embedded content, such as videos or animations. Although Docling can replace some non-text content with placeholders during ingestion, some embedded content might cause errors.
- Make sure your Podman/Docker VM has sufficient memory and temporary storage for the ingestion tasks. The minimum recommendation is 8 GB of RAM and at least 50 GB of free disk space, but the exact requirements depend on the size and complexity of your documents. If you regularly ingest large files, more RAM and space are recommended.
- If OCR ingestion fails due to OCR missing, see OCR ingestion fails (easyocr not installed).

Timeouts when ingesting many files or very large files

Ingesting very large PDFs (more than 300 pages) and folders with many documents can take 30 minutes or more.

Make sure your container VM has sufficient memory for processing large files and folders.

If you experience timeouts during ingestion, edit the following environment variables in your OpenRAG `.env` file:

- `LANGFLOW_TIMEOUT`
- `LANGFLOW_CONNECT_TIMEOUT`
- `INGESTION_TIMEOUT`

- `UPLOAD_BATCH_SIZE`
- `MAX_WORKERS`
- `LANGFLOW_WORKERS`
- `DOCLING_WORKERS`

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 `~/.cache/uv`.

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

1. If the TUI is open, press `q` to exit the TUI.
2. Clear the `uv` cache:

   ```
   uv cache clean
   ```

   Or clear only the OpenRAG cache:

   ```
   uv cache clean openrag
   ```

3. Invoke OpenRAG to restart the TUI:

   ```
   uvx openrag
   ```

4. Click **Launch OpenRAG**, 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`.

Problems when referencing documents in chat

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 you applied a filter to the chat, make sure the expected documents aren't excluded by the filter settings. You can test this by applying the filter when you browse the knowledge base. If the filter excludes any documents, the agent cannot access those documents. Be aware that some settings create dynamic filters that don't always produce the same results, such as a **Search query** combined with a low **Response limit**.

If the document chunks have missing, incorrect, or unexpected text, you must delete the documents from your knowledge base, modify the ingestion settings or the documents themselves, and then reingest the documents. 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.

Chat issues

The following issues can occur when using the OpenRAG **Chat** feature.

Documents seem to be missing or misinterpreted

In the **Chat**, click **Function Call: search_documents (tool_call)** to view the log of tool calls made by the agent. This can shows you how the agent used particular tools and the reasoning.

Click **Knowledge** to confirm that the documents are present in the OpenRAG OpenSearch knowledge base, 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 settings or modify and reupload the document.

See also Document ingestion or similarity search issues.

Service is suddenly unavailable when it was working previously

First, verify that the container VM and the OpenRAG services are running and healthy.

Second, make sure there are no issues with the flow configuration. If you edited the **OpenRAG OpenSearch Agent** flow, use the **Restore flow** option to revert the flow to its original configuration.

If you want to preserve your customizations, you can export the flow before restoring the flow.

JSON-formatted responses

If a model returns JSON-formatted output instead of natural language responses, the model might not be designed for agentic reasoning and RAG tasks.

Try a different model.

For more information, see Model availability and performance issues.

Frequent hallucinations

If the model seems to hallucinate frequently, the model might be too small or poorly suited to RAG tasks.

Try a different model.

For more information, see Model availability and performance issues.

Responses are good but could be better

If your model is compatible with OpenRAG, and it isn't exhibiting any obvious errors, then you can use RAG best practices to refine the response quality, such as knowledge filters and prompt engineering.

Cannot detect GPU with Fedora or local Ollama

If your machine has GPU support, but OpenRAG is having problems detecting or using the GPU, you might see errors like the following:

```
Auto-detected mode as 'legacy'
nvidia-container-cli: ldcache error
error running createRuntime hook
unrecognized runtime 'crun'
OCI runtime error
```

To troubleshoot these issues, do the following:

- Make sure your environment has a compatible version of the NVIDIA Container Toolkit.
- If you are using a local Ollama deployment, Ollama must run in a container with GPU support.

The following steps explain how to troubleshoot these issues with Fedora 43, Podman, local Ollama, and a machine that has an NVIDIA GPU:

1. Make sure your NVIDIA driver is updated to the latest version.
2. Remove old toolkit packages:

   ```
   sudo dnf5 remove -y nvidia-container-toolkit nvidia-container-toolkit-base libnvidia-container*
   ```

   This command doesn't remove the NVIDIA driver.
3. Get the NVIDIA Container Toolkit package repository:

   ```
   sudo curl -s -o /etc/yum.repos.d/nvidia-container-toolkit.repo \
   https://nvidia.github.io/libnvidia-container/stable/rpm/nvidia-container-toolkit.repo
   ```

Version 1.14 or later is required.

4. Refresh the package cache to recognize the new repository:

```
sudo dnf5 clean expire-cache
sudo dnf5 update
```

5. Install the new toolkit:

```
sudo dnf5 install -y nvidia-container-toolkit
```

This command installs the following:

- `libnvidia-container` version 1.14 or later
- `nvidia-container-toolkit-base` version 1.14 or later
- `nvidia-container-toolkit` version 1.14 or later

These versions support Fedora 43, Podman, and `crun`.

6. Configure the runtime for Podman (`crun`):

```
sudo nvidia-ctk runtime configure --runtime=crun
```

If this command fails, the make sure you installed version 1.14 or later of the NVIDIA Container Toolkit, which supports `crun`.

7. Fix the dynamic loader cache to prevent `ldconfig` errors:

```
sudo ldconfig
```

8. Restart the Podman socket:

```
systemctl --user enable --now podman.socket
```

9. Test the NVIDIA hook, and make sure the output doesn't include `Auto-detected mode as 'legacy'`. The output must be empty.

```
/usr/bin/nvidia-container-runtime-hook prestart
```

10. Test the GPU with Podman:

```
podman run --rm \
--hooks-dir=/usr/share/containers/oci/hooks.d \
--device nvidia.com/gpu=all \
nvidia/cuda:12.3.0-base-ubuntu22.04 nvidia-smi
```

Make sure `nvidia-smi` prints your GPU information instead of an error.

The `--rm` argument automatically removes the container after the command exits.

11. Run Ollama in a GPU-enabled container:

```
podman run -d \
--name ollama \
--security-opt=label=disable \
--hooks-dir=/usr/share/containers/oci/hooks.d \
--device nvidia.com/gpu=all \
-p 11434:11434 \
-v $HOME/ollama:/root/.ollama \
docker.io/ollama/ollama:latest
```

12. Start the TUI, and then start OpenRAG with GPU mode enabled. Or, for self-managed deployments, deploy OpenRAG with the `docker-compose-gpu.yml` file.