What is OpenRAG?

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

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

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

- 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.

- Docling: Docling simplifies document processing, parsing diverse formats — including advanced PDF understanding — and providing seamless integrations with the gen AI ecosystem.

OpenRAG builds on Langflow's familiar interface while adding OpenSearch for vector storage and Docling for simplified document parsing, with opinionated flows that serve as ready-to-use recipes for ingestion, retrieval, and generation from popular sources like Google Drive, OneDrive, and Sharepoint.

What's more, every part of the stack is swappable. Write your own custom components in Python, try different language models, and customize your flows to build an agentic RAG system.

Ready to get started? Install OpenRAG and then run the Quickstart to create a powerful RAG pipeline.

OpenRAG architecture

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

The **OpenRAG Backend** is the central orchestration service that coordinates all other components.

Langflow provides a visual workflow engine for building AI agents, and connects to **OpenSearch** for vector storage and retrieval.

Docling Serve is a local document processing service managed by the **OpenRAG Backend**.

Third Party Services like **Google Drive** connect to the **OpenRAG Backend** through OAuth authentication, allowing synchronication of cloud storage with the OpenSearch knowledge base.

The **OpenRAG Frontend** provides the user interface for interacting with the system.

Performance expectations

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 on servers. Throughput scales with CPU cores, memory, storage speed, and configuration choices such as embedding model, chunk size and overlap, and concurrency.

This test returned 12 errors (approximately 1.1%). 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
```

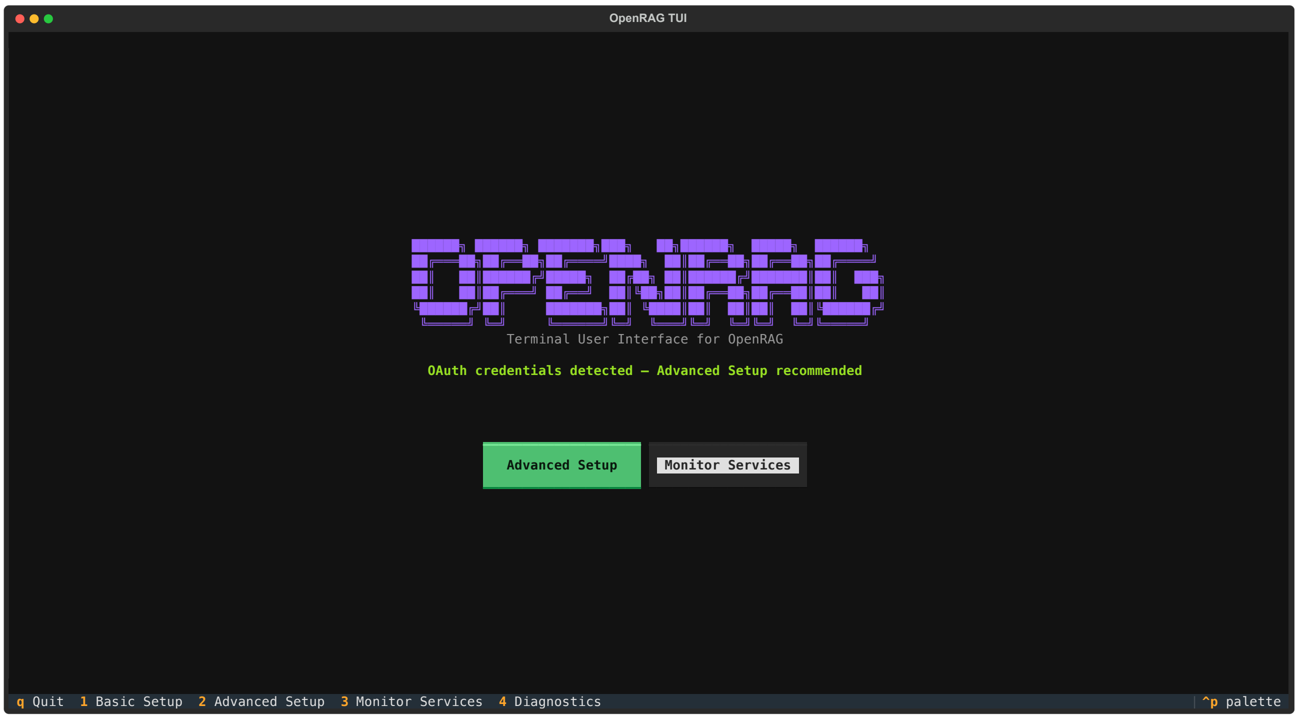
Elapsed time: ~42 minutes 15 seconds (2,535 seconds)

Throughput: ~2.4 documents/second

Install OpenRAG with TUI

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

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



```
Terminal User Interface for OpenRAG
OAuth credentials detected – Advanced Setup recommended
```

OpenRAG TUI
OPENRAG
Advanced Setup
Monitor Services
q Quit 1 Basic Setup 2 Advanced Setup 3 Monitor Services 4 Diagnostics
^p palette

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

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

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

Prerequisites

- Install Python Version 3.10 to 3.13
- Install uv
- Install Podman (recommended) or Docker

- Install Docker Compose. If using Podman, use podman-compose or alias Docker compose commands to Podman commands.
- Create an OpenAI API key. This key is **required** to start OpenRAG, but you can choose a different model provider during Application Onboarding.
- Optional: Install GPU support with an NVIDIA GPU, CUDA support, and compatible NVIDIA drivers on the OpenRAG host machine. If you don't have GPU capabilities, OpenRAG provides an alternate CPU-only deployment.

Install OpenRAG

> **WINDOWS USERS**
>
> To use OpenRAG on Windows, use WSL (Windows Subsystem for Linux).

To set up a project and install OpenRAG as a dependency, do the following:

1. Create a new project with a virtual environment using `uv init`.

   ```
   uv init YOUR_PROJECT_NAME
   cd YOUR_PROJECT_NAME
   ```

   The `(venv)` prompt doesn't change, but `uv` commands will automatically use the project's virtual environment. For more information on virtual environments, see the uv documentation.

2. Add OpenRAG to your project.

   ```
   uv add openrag
   ```

   To add a specific version of OpenRAG:

   ```
   uv add openrag==0.1.25
   ```

3. Start the OpenRAG TUI.

```
uv run openrag
```

Install a local wheel

If you downloaded the OpenRAG wheel to your local machine, follow these steps:

1. Add the wheel to your project's virtual environment.

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

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

   For example, if your `.whl` file is in the `~/Downloads` directory:

   ```
   uv add ~/Downloads/openrag-0.1.8-py3-none-any.whl
   ```

2. Run OpenRAG.

   ```
   uv run openrag
   ```

3. Continue with Set up OpenRAG with the TUI.

Set up OpenRAG with the TUI

The TUI creates a `.env` file in your OpenRAG directory root and starts OpenRAG. If the TUI detects a `.env` file in the OpenRAG root directory, it sources any variables from the `.env` file. If the TUI detects OAuth credentials, it enforces the **Advanced Setup** path.

Basic setup

Basic Setup generates all of the required values for OpenRAG except the OpenAI API key. **Basic Setup** does not set up OAuth connections for ingestion from cloud providers. For OAuth setup, use **Advanced Setup**. For information about the difference between basic (no auth) and OAuth in OpenRAG, see Authentication and document access.

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

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

   ```
   Services started successfully
   Command completed successfully
   ```
6. To open the OpenRAG application, click **Open App**.
7. Continue with Application Onboarding.

Advanced setup

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

   The OpenSearch password is required. The Langflow admin password is optional. If no Langflow admin password is generated, Langflow runs in autologin mode with no password required.
3. Paste your OpenAI API key in the OpenAI API key field.
4. Add your client and secret values for Google or Microsoft OAuth. These values can be found with your OAuth provider. For more information, see the Google OAuth client or Microsoft Graph OAuth client documentation.

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

6. Click **Save Configuration**.

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

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

8. To open the OpenRAG application, click **Open App**. You are presented with your provider's OAuth sign-in screen. After sign-in, you are redirected to the redirect URI.

   Two additional variables are available for Advanced Setup:

   The `LANGFLOW_PUBLIC_URL` controls where the Langflow web interface can be accessed. This is where users interact with their flows in a browser.

   The `WEBHOOK_BASE_URL` controls where the endpoint for `/connectors/CONNECTOR_TYPE/webhook` will be available. This connection enables real-time document synchronization with external services. Supported webhook endpoints:

   - Google Drive: `/connectors/google_drive/webhook`
   - OneDrive: `/connectors/onedrive/webhook`
   - SharePoint: `/connectors/sharepoint/webhook`

9. Continue with Application Onboarding.

Application onboarding

The first time you start OpenRAG, whether using the TUI or a `.env` file, it's recommended that you complete application onboarding.

To skip onboarding, click **Skip onboarding**.

Values from onboarding can be changed later in the OpenRAG **Settings** page.

Choose one LLM provider and complete only those steps:

OpenAI

1. Enable **Get API key from environment variable** to automatically enter your key from the TUI-generated `.env` file. Alternatively, paste an OpenAI API key into the field.
2. Under **Advanced settings**, select your **Embedding Model** and **Language Model**.
3. To load 2 sample PDFs, enable **Sample dataset**. This is recommended, but not required.
4. Click **Complete**.
5. To complete the onboarding tasks, click **What is OpenRAG**, and then click **Add a Document**.
6. Continue with the Quickstart.

IBM watsonx.ai

1. Complete the fields for **watsonx.ai API Endpoint**, **IBM Project ID**, and **IBM API key**. These values are found in your IBM watsonx deployment.
2. Under **Advanced settings**, select your **Embedding Model** and **Language Model**.
3. To load 2 sample PDFs, enable **Sample dataset**. This is recommended, but not required.
4. Click **Complete**.
5. To complete the onboarding tasks, click **What is OpenRAG**, and then click **Add a Document**.
6. Continue with the Quickstart.

Ollama

TIP

Ollama is not included with OpenRAG. To install Ollama, see the Ollama documentation.

1. Enter your Ollama server's base URL address. The default Ollama server address is `http://localhost:11434`. OpenRAG automatically transforms `localhost` to

access services outside of the container, and sends a test connection to your Ollama server to confirm connectivity.
2. Select the **Embedding Model** and **Language Model** your Ollama server is running. OpenRAG retrieves the available models from your Ollama server.
3. To load 2 sample PDFs, enable **Sample dataset**. This is recommended, but not required.
4. Click **Complete**.
5. To complete the onboarding tasks, click **What is OpenRAG**, and then click **Add a Document**.
6. Continue with the Quickstart.

Close the OpenRAG TUI

To close the OpenRAG TUI, press `q`. The OpenRAG containers will continue to be served until the containers are stopped. For more information, see Manage OpenRAG containers with the TUI .

To start the TUI again, run `uv run openrag`.

Manage OpenRAG containers with the TUI

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

Start all services

Click **Start All Services** to start the OpenRAG containers. The TUI 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. The TUI then pulls the images and deploys the containers with the following command.

```
docker compose up -d
```

If images are missing, the TUI runs `docker compose pull`, then runs `docker compose up -d`.

Status

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

To view streaming logs, select the container you want to view, and press `l`. To copy your logs, click **Copy to Clipboard**.

To **upgrade** your containers, click **Upgrade**. **Upgrade** runs `docker compose pull` and then `docker compose up -d --force-recreate`. The first command pulls the latest images of OpenRAG. The second command recreates the containers with your data persisted.

To **reset** your containers, click **Reset**. Reset gives you a completely fresh start. Reset deletes all of your data, including OpenSearch data, uploaded documents, and authentication. **Reset** runs two commands. It first stops and removes all containers, volumes, and local images.

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

When the first command is complete, OpenRAG removes any additional Docker objects with `prune`.

```
docker system prune -f
```

Native services status

A *native service* in OpenRAG refers to a service run locally on your machine, and not within a container. The `docling serve` process is a native service in OpenRAG, because it's a document processing service that is run on your local machine, and controlled separately from the containers.

To start or stop `docling serve` or any other native services, in the TUI Status menu, click **Stop** or **Restart**.

To view the status, port, or PID of a native service, in the TUI main menu, click Status.

Diagnostics

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

Install OpenRAG containers

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

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

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

Prerequisites

- Install Python Version 3.10 to 3.13
- Install uv
- Install Podman (recommended) or Docker
- Install Docker Compose. If using Podman, use podman-compose or alias Docker compose commands to Podman commands.
- Create an OpenAI API key. This key is **required** to start OpenRAG, but you can choose a different model provider during Application Onboarding.
- Optional: Install GPU support with an NVIDIA GPU, CUDA support, and compatible NVIDIA drivers on the OpenRAG host machine. If you don't have GPU capabilities, OpenRAG provides an alternate CPU-only deployment.

Install OpenRAG with Docker Compose

To install OpenRAG with Docker Compose, do the following:

1. Clone the OpenRAG repository.

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

2. Install dependencies.

```
uv sync
```

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

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

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

   ```
   touch .env
   ```

4. The Docker Compose files are populated with the values from your `.env` file. The following values must be set:

   ```
   OPENSEARCH_PASSWORD=your_secure_password
   OPENAI_API_KEY=your_openai_api_key
   LANGFLOW_SECRET_KEY=your_secret_key
   ```

   `OPENSEARCH_PASSWORD` can be automatically generated when using the TUI, but for a Docker Compose installation, you can set it manually instead. To generate an OpenSearch admin password, see the OpenSearch documentation.

   The `OPENAI_API_KEY` is found in your OpenAI account.

   `LANGFLOW_SECRET_KEY` is automatically generated when using the TUI, and Langflow will also auto-generate it if not set. For more information, see the Langflow documentation.

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

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

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

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

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

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

6. Confirm `docling serve` is running.

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

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

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

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

For GPU support (docker-compose.yml):

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

For CPU-only (docker-compose-cpu.yml):

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

The OpenRAG Docker Compose file starts five containers:

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

8. Verify installation by confirming all services are running.

   ```
   docker compose ps
   ```

   You can now access OpenRAG at the following endpoints:

   - **Frontend**: http://localhost:3000
   - **Backend API**: http://localhost:8000
   - **Langflow**: http://localhost:7860
9. Continue with Application Onboarding.

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

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

Application onboarding

The first time you start OpenRAG, whether using the TUI or a `.env` file, it's recommended that you complete application onboarding.

To skip onboarding, click **Skip onboarding**.

Values from onboarding can be changed later in the OpenRAG **Settings** page.

Choose one LLM provider and complete only those steps:

OpenAI

1. Enable **Get API key from environment variable** to automatically enter your key from the TUI-generated `.env` file. Alternatively, paste an OpenAI API key into the field.
2. Under **Advanced settings**, select your **Embedding Model** and **Language Model**.
3. To load 2 sample PDFs, enable **Sample dataset**. This is recommended, but not required.
4. Click **Complete**.
5. To complete the onboarding tasks, click **What is OpenRAG**, and then click **Add a Document**.
6. Continue with the Quickstart.

IBM watsonx.ai

1. Complete the fields for **watsonx.ai API Endpoint**, **IBM Project ID**, and **IBM API key**. These values are found in your IBM watsonx deployment.
2. Under **Advanced settings**, select your **Embedding Model** and **Language Model**.
3. To load 2 sample PDFs, enable **Sample dataset**. This is recommended, but not required.
4. Click **Complete**.
5. To complete the onboarding tasks, click **What is OpenRAG**, and then click **Add a Document**.
6. Continue with the Quickstart.

Ollama

> **TIP**
>
> Ollama is not included with OpenRAG. To install Ollama, see the Ollama documentation.

1. Enter your Ollama server's base URL address. The default Ollama server address is `http://localhost:11434`. OpenRAG automatically transforms `localhost` to

access services outside of the container, and sends a test connection to your Ollama server to confirm connectivity.
2. Select the **Embedding Model** and **Language Model** your Ollama server is running. OpenRAG retrieves the available models from your Ollama server.
3. To load 2 sample PDFs, enable **Sample dataset**. This is recommended, but not required.
4. Click **Complete**.
5. To complete the onboarding tasks, click **What is OpenRAG**, and then click **Add a Document**.
6. Continue with the Quickstart.

Container management commands

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

Upgrade containers

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

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

Rebuild containers (destructive)

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

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

Remove all containers and data (destructive)

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

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

Quickstart

Get started with OpenRAG by loading your knowledge, swapping out your language model, and then chatting with the Langflow API.

Prerequisites

- Install and start OpenRAG with the TUI or Docker

Load and chat with your own documents

1. In OpenRAG, click **Chat**. The chat is powered by the OpenRAG OpenSearch Agent. For more information, see Langflow in OpenRAG.
2. Ask `What documents are available to you?` The agent responds with a message summarizing the documents that OpenRAG loads by default. Knowledge is stored in OpenSearch. For more information, see OpenSearch in OpenRAG.
3. To confirm the agent is correct about the default knowledge, click **Knowledge**. The **Knowledge** page lists the documents OpenRAG has ingested into the OpenSearch vector database. Click on a document to display the chunks derived from splitting the default documents into the OpenSearch vector database.
4. To add documents to your knowledge base, click **Add Knowledge**.
   - Select **File** to add a single file from your local machine.
   - Select **Folder** to process an entire folder of documents from your local machine. The default directory is `/documents` in your OpenRAG directory.
   - Select your cloud storage provider to add knowledge from an OAuth-connected storage provider. For more information, see OAuth ingestion.
5. Return to the Chat window and ask a question about your loaded data. For example, with a manual about a PC tablet loaded, ask `How do I connect this device to WiFi?` The agent responds with a message indicating it now has your knowledge as context for answering questions.
6. Click **Function Call: search_documents (tool_call)**. This log describes how the agent uses tools. This is helpful for troubleshooting when the agent isn't responding as expected.

Swap out the language model to modify agent behavior

To modify the knowledge ingestion or Agent behavior, click **Settings**.

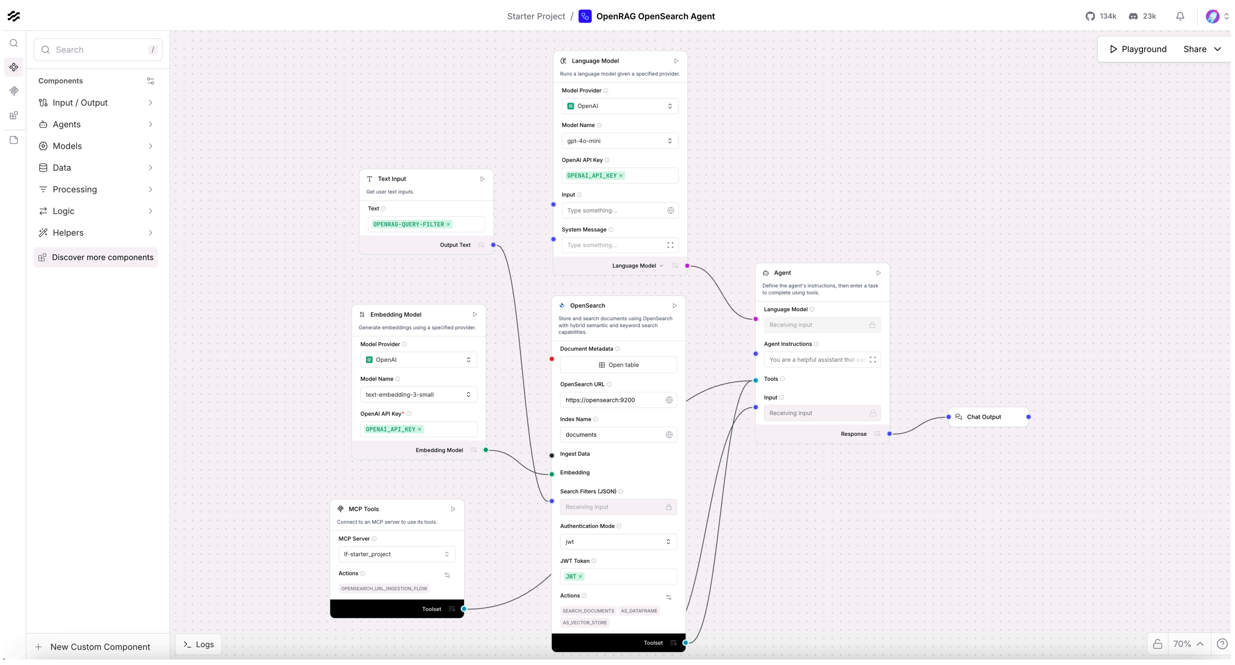
In this example, you'll try a different LLM to demonstrate how the Agent's response changes.

1. To edit the Agent's behavior, click **Edit in Langflow**. You can more quickly access the **Language Model** and **Agent Instructions** fields in this page, but for illustration purposes, navigate to the Langflow visual builder. To revert the flow to its initial state, click **Restore flow**.

2. OpenRAG warns you that you're entering Langflow. Click **Proceed**.

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

   The OpenRAG OpenSearch Agent flow appears in a new browser window.


Starter Project / OpenRAG OpenSearch Agent
134k
23k
Playground
Share
Search
Components
Input / Output
Agents
Models
Data
Processing
Logic
Helpers
Discover more components
New Custom Component
Logs
70%
Language Model
Runs a language model given a specified provider.
Text Input
Get user text inputs.
Output Text
Embedding Model
Generate embeddings using a specified provider.
OpenSearch
Store and search documents using OpenSearch with hybrid semantic and keyword search capabilities.
Agent
Define the agent's instructions, then enter a task to complete using tools.
MCP Tools
Connect to an MCP server to use its tools.
Chat Output
Response
Toolset
Ingest Data
Embedding
Open table

Model Provider: OpenAI
Model Name: gpt-4o-mini
OpenAI API Key: OPENAI_API_KEY
Input: Type something...
System Message: Type something...
Text: OPENRAG-QUERY-FILTER
Model Provider: OpenAI
Model Name: text-embedding-3-small
OpenAI API Key*: OPENAI_API_KEY
Document Metadata
OpenSearch URL: https://opensearch:9200
Index Name: documents
Search Filters (JSON): Receiving input
Authentication Mode: jwt
JWT Token: JWT
Actions: SEARCH_DOCUMENTS AS_DATAFRAME AS_VECTOR_STORE
Language Model: Receiving input
Agent Instructions: You are a helpful assistant that can
Tools
Input: Receiving input
MCP Server: lf-starter_project
Actions: OPENSEARCH_URL_INGESTION_FLOW

3. Find the **Language Model** component, and then change the **Model Name** field to a different OpenAI model.

4. Save your flow with `Command+S` (Mac) or `Ctrl+S` (Windows).

5. Return to the OpenRAG browser window, and start a new conversation by clicking + in the **Conversations** tab.

6. Ask the same question you asked before to see how the response differs between models.

Integrate OpenRAG into your application

Langflow in OpenRAG includes pre-built flows that you can integrate into your applications using the Langflow API.

The Langflow API accepts Python, TypeScript, or curl requests to run flows and get responses. You can use these flows as-is or modify them to better suit your needs.

In this section, you'll run the OpenRAG OpenSearch Agent flow and get a response using the API.

1. To navigate to the OpenRAG OpenSearch Agent flow in Langflow, click **Settings**, and then click **Edit in Langflow** in the OpenRAG OpenSearch Agent flow.

2. Create a Langflow API key.

   A Langflow API key is a user-specific token you can use with Langflow. It is **only** used for sending requests to the Langflow server. It does **not** access OpenRAG.

   To create a Langflow API key, do the following:

   i. Open Langflow, click your user icon, 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.
3. Langflow includes code snippets for the request to the Langflow API. To retrieve the code snippet, click **Share**, and then click **API access**.

   The default code in the API access pane constructs a request with the Langflow server `url`, `headers`, and a `payload` of request data. The code snippets automatically include the `LANGFLOW_SERVER_ADDRESS` and `FLOW_ID` values for the flow.

   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 the snippet, paste it in a script file, and then run the script to send the request. If you are using the curl snippet, you can run the command 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.

To further explore the API, see:

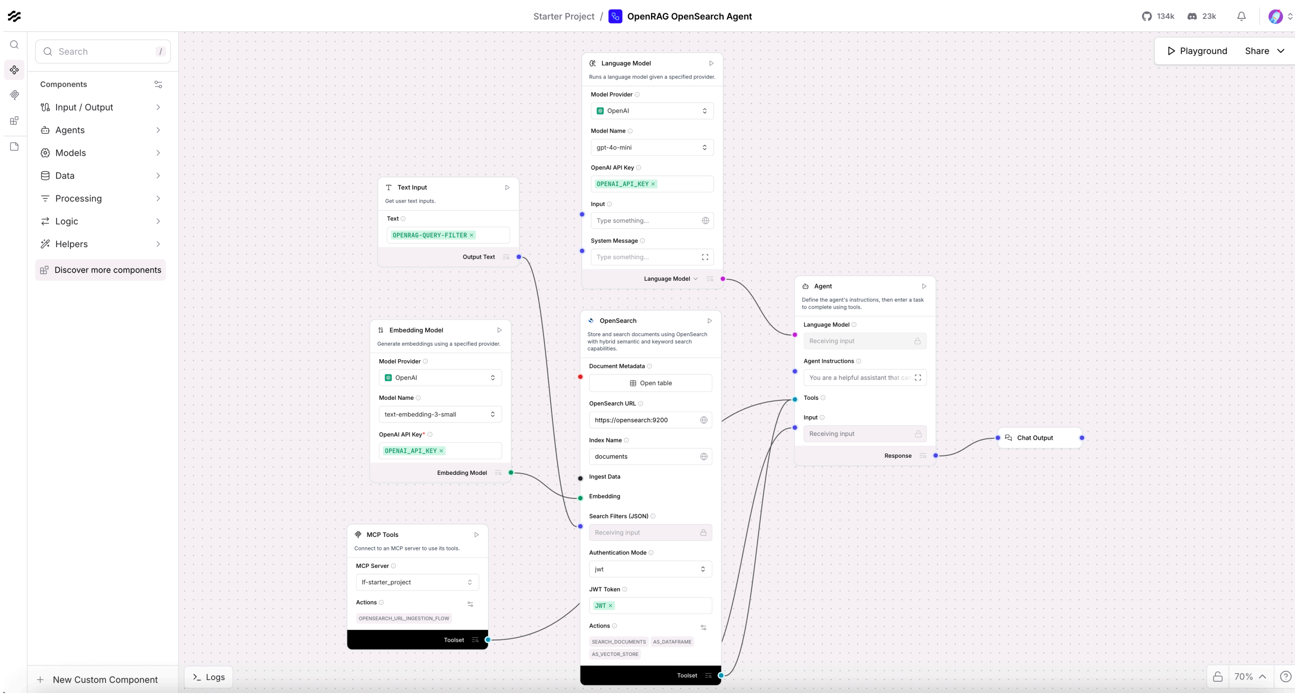
- The Langflow Quickstart extends this example with extracting fields from the response.
- Get started with the Langflow API

Langflow in OpenRAG

OpenRAG leverages Langflow's Agent component to power the OpenRAG OpenSearch Agent flow.

Flows in Langflow are functional representations of application workflows, with multiple component nodes connected as single steps in a workflow.

In the OpenRAG OpenSearch Agent flow, components like the Langflow **Agent** component and **OpenSearch** component are connected to intelligently chat with your knowledge by embedding your query, comparing it the vector database embeddings, and generating a response with the LLM.



Text Input
Get user text inputs.
Text: OPENRAG-QUERY-FILTER
Output Text

Language Model
Runs a language model given a specified provider.
Model Provider: OpenAI
Model Name: gpt-4o-mini
OpenAI API Key: OPENAI_API_KEY
Input: Type something...
System Message: Type something...
Language Model

Embedding Model
Generate embeddings using a specified provider.
Model Provider: OpenAI
Model Name: text-embedding-3-small
OpenAI API Key*: OPENAI_API_KEY
Embedding Model

OpenSearch
Store and search documents using OpenSearch with hybrid semantic and keyword search capabilities.
Document Metadata: Open table
OpenSearch URL: https://opensearch:9200
Index Name: documents
Ingest Data
Embedding
Search Filters (JSON): Receiving input
Authentication Mode: jwt
JWT Token: JWT
Actions: SEARCH_DOCUMENTS AS_DATAFRAME AS_VECTOR_STORE
Toolset

MCP Tools
Connect to an MCP server to use its tools.
MCP Server: lf-starter_project
Actions: OPENSEARCH_URL_INGESTION_FLOW
Toolset

Agent
Define the agent's instructions, then enter a task to complete using tools.
Language Model: Receiving input
Agent Instructions: You are a helpful assistant that can
Tools
Input: Receiving input
Response

Chat Output

The Agent component shines here in its ability to make decisions on not only what query should be sent, but when a query is necessary to solve the problem at hand.

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.

Use the OpenRAG OpenSearch Agent flow

If you've chatted with your knowledge in OpenRAG, you've already experienced the OpenRAG OpenSearch Agent chat flow. To switch OpenRAG over to the Langflow visual editor and view the OpenRAG OpenSearch Agentflow, click **Settings**, and then click **Edit in Langflow**. This flow contains eight components connected together to chat with your data:

- The **Agent** component orchestrates the entire flow by deciding when to search the knowledge base, how to formulate search queries, and how to combine retrieved information with the user's question to generate a comprehensive response. The **Agent** behaves according to the prompt in the **Agent Instructions** field.
- The **Chat Input** component is connected to the Agent component's Input port. This allows to flow to be triggered by an incoming prompt from a user or application.
- The **OpenSearch** component is connected to the Agent component's Tools port. The agent may not use this database for every request; the agent only uses this connection if it decides the knowledge can help respond to the prompt.
- The **Language Model** component is connected to the Agent component's Language Model port. The agent uses the connected LLM to reason through the request sent through Chat Input.

- The **Embedding Model** component is connected to the OpenSearch component's Embedding port. This component converts text queries into vector representations that are compared with document embeddings stored in OpenSearch for semantic similarity matching. This gives your Agent's queries context.
- The **Text Input** component is populated with the global variable `OPENRAG-QUERY-FILTER`. This filter is the Knowledge filter, and filters which knowledge sources to search through.
- The **Agent** component's Output port is connected to the **Chat Output** component, which returns the final response to the user or application.
- An **MCP Tools** component is connected to the Agent's **Tools** port. This component calls the OpenSearch URL Ingestion flow, which Langflow uses as an MCP server to fetch content from URLs and store in OpenSearch.

All flows included with OpenRAG are designed to be modular, performant, and provider-agnostic. To modify a flow, click **Settings**, and click **Edit in Langflow**. OpenRAG's visual editor is based on the Langflow visual editor, so you can edit your flows to match your specific use case.

For an example of changing out the agent's language model in OpenRAG, see the Quickstart.

To restore the flow to its initial state, in OpenRAG, click **Settings**, and then click **Restore Flow**. OpenRAG warns you that this discards all custom settings. Click **Restore** to restore the flow.

Additional Langflow functionality

Langflow includes features beyond Agents to help you integrate OpenRAG into your application, and all Langflow features are included in OpenRAG.

- Langflow can serve your flows as an MCP server, or consume other MCP servers as an MCP client. Get started with the MCP tutorial.

- If you don't see the component you need, extend Langflow's functionality by creating custom Python components.

- Langflow offers component bundles to integrate with many popular vector stores, AI/ML providers, and search APIs.

OpenSearch in OpenRAG

OpenRAG uses OpenSearch for its vector-backed knowledge store. This is a specialized database for storing and retrieving embeddings, which helps your Agent efficiently find relevant information. OpenSearch provides powerful hybrid search capabilities with enterprise-grade security and multi-tenancy support.

Authentication and document access

OpenRAG supports two authentication modes based on how you install OpenRAG, and which mode you choose affects document access.

No-auth mode (Basic Setup): This mode uses a single anonymous JWT token for OpenSearch authentication, so documents uploaded to the `documents` index by one user are visible to all other users on the OpenRAG server.

OAuth mode (Advanced Setup): Each OpenRAG user is granted a JWT token, and each document is tagged with user ownership. Documents are filtered by user ownership, ensuring users only see documents they uploaded or have access to.

Ingest knowledge

OpenRAG supports knowledge ingestion through direct file uploads and OAuth connectors. To configure the knowledge ingestion pipeline parameters, see Docling Ingestion.

Direct file ingestion

The **Knowledge Ingest** flow uses Langflow's **File** component to split and embed files loaded from your local machine into the OpenSearch database.

The default path to your local folder is mounted from the `./documents` folder in your OpenRAG project directory to the `/app/documents/` directory inside the Docker container. Files added to the host or the container will be visible in both locations. To configure this location, modify the **Documents Paths** variable in either the TUI's Advanced Setup menu or in the `.env` used by Docker Compose.

To load and process a single file from the mapped location, click **Add Knowledge**, and then click 🗋 **File**. The file is loaded into your OpenSearch database, and appears in the

Knowledge page.

To load and process a directory from the mapped location, click **Add Knowledge**, and then click 🗀 **Folder**. The files are loaded into your OpenSearch database, and appear in the Knowledge page.

To add files directly to a chat session, click + in the chat input and select the files you want to include. Files added this way are processed and made available to the agent for the current conversation, and are not permanently added to the knowledge base.

Ingest files through OAuth connectors

OpenRAG supports Google Drive, OneDrive, and Sharepoint as OAuth connectors for seamless document synchronization.

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

Before users can connect their cloud storage accounts, you must configure OAuth credentials in OpenRAG. This requires registering OpenRAG as an OAuth application with a cloud provider and obtaining client ID and secret keys for each service you want to support.

To add an OAuth connector to OpenRAG, do the following. This example uses Google OAuth. If you wish to use another provider, add the secrets to another provider.

TUI installation

1. If OpenRAG is running, stop it with **Status** > **Stop Services**.
2. Click **Advanced Setup**.
3. Add the OAuth provider's client and secret key in the Advanced Setup menu.
4. Click **Save Configuration**. The TUI generates a new `.env` file with your OAuth values.
5. Click **Start Container Services**.

.env file installation

1. Stop the Docker deployment.
2. Add the OAuth provider's client and secret key in the `.env` file for Docker Compose.

   ```
   GOOGLE_OAUTH_CLIENT_ID='YOUR_OAUTH_CLIENT_ID'
   GOOGLE_OAUTH_CLIENT_SECRET='YOUR_OAUTH_CLIENT_SECRET'
   ```

3. Save your `.env` file.
4. Start the Docker deployment.

The OpenRAG frontend at `http://localhost:3000` now redirects to an OAuth callback login page for your OAuth provider. A successful authentication opens OpenRAG with the required scopes for your connected storage.

To add knowledge from an OAuth-connected storage provider, do the following:

1. Click **Add Knowledge**, and then select the storage provider, for example, **Google Drive**. The **Add Cloud Knowledge** page opens.
2. To add files or folders from the connected storage, click **Add Files**. Select the files or folders you want and click **Select**. You can select multiple files.
3. When your files are selected, click **Ingest Files**. The ingestion process may take some time, depending on the size of your documents.
4. When ingestion is complete, your documents are available in the Knowledge screen.

Explore knowledge

The **Knowledge** page lists the documents OpenRAG has ingested into the OpenSearch vector database's `documents` index.

To explore your current knowledge, click **Knowledge**. Click on a document to display the chunks derived from splitting the default documents into the vector database.

Documents are processed with the default **Knowledge Ingest** flow, so if you want to split your documents differently, edit the **Knowledge Ingest** flow.

All flows included with OpenRAG are designed to be modular, performant, and provider-agnostic. To modify a flow, click **Settings**, and click **Edit in Langflow**. OpenRAG's visual editor is based on the Langflow visual editor, so you can edit your flows to match your specific use case.

Create knowledge filters

OpenRAG includes a knowledge filter system for organizing and managing document collections. Knowledge filters are saved search configurations that allow you to create custom views of your document collection. They store search queries, filter criteria, and display settings that can be reused across different parts of OpenRAG.

Knowledge filters help agents work more efficiently with large document collections by focusing their context within relevant documents sets.

To create a knowledge filter, do the following:

1. Click **Knowledge**, and then click + **Knowledge Filters**. The **Knowledge Filter** pane appears.

2. Enter a **Name** and **Description**, and then click **Create Filter**. A new filter is created with default settings that match all documents.

3. To modify the filter, click **Knowledge**, and then click your new filter to edit it in the **Knowledge Filter** pane.

   The following filter options are configurable.

   - **Search Query**: Enter text for semantic search, such as "financial reports from Q4".
   - **Data Sources**: Select specific data sources or folders to include.
   - **Document Types**: Filter by file type.
   - **Owners**: Filter by who uploaded the documents.
   - **Connectors**: Filter by connector types, such as local upload or Google Drive.
   - **Response Limit**: Set maximum number of results. The default is `10`.
   - **Score Threshold**: Set minimum relevance score. The default score is `0`.
4. When you're done editing the filter, click **Update Filter**.

5. To apply the filter to OpenRAG globally, click **Knowledge**, and then select the filter to apply. One filter can be enabled at a time.

   To apply the filter to a single chat session, in the **Chat** window, click , and then select the filter to apply.

To delete the filter, in the **Knowledge Filter** pane, click **Delete Filter**.

OpenRAG default configuration

OpenRAG automatically detects and configures the correct vector dimensions for embedding models, ensuring optimal search performance and compatibility.

The complete list of supported models is available at `models_service.py` in the OpenRAG repository.

You can use custom embedding models by specifying them in your configuration.

If you use an unknown embedding model, OpenRAG will automatically fall back to `1536` dimensions and log a warning. The system will continue to work, but search quality may be affected if the actual model dimensions differ from `1536`.

The default embedding dimension is `1536` and the default model is `text-embedding-3-small`.

For models with known vector dimensions, see `settings.py` in the OpenRAG repository.

Docling in OpenRAG

OpenRAG uses Docling for document ingestion. More specifically, OpenRAG uses Docling Serve, which starts a `docling serve` process on your local machine and runs Docling ingestion through an API service.

Docling ingests documents from your local machine or OAuth connectors, splits them into chunks, and stores them as separate, structured documents in the OpenSearch `documents` index.

OpenRAG chose Docling for its support for a wide variety of file formats, high performance, and advanced understanding of tables and images.

To modify OpenRAG's ingestion settings, including the Docling settings and ingestion flows, click 2" aria-hidden="true"/> **Settings**.

Knowledge ingestion settings

These settings configure the Docling ingestion parameters.

OpenRAG will warn you if `docling serve` is not running. To start or stop `docling serve` or any other native services, in the TUI main menu, click **Start Native Services** or **Stop Native Services**.

Embedding model determines which AI model is used to create vector embeddings. The default is the OpenAI `text-embedding-3-small` model.

Chunk size determines how large each text chunk is in number of characters. Larger chunks yield more context per chunk, but may include irrelevant information. Smaller chunks yield more precise semantic search, but may lack context. The default value of `1000` characters provides a good starting point that balances these considerations.

Chunk overlap controls the number of characters that overlap over chunk boundaries. Use larger overlap values for documents where context is most important, and use smaller overlap values for simpler documents, or when optimization is most important. The default value of 200 characters of overlap with a chunk size of 1000 (20% overlap) is suitable for general use cases. Decrease the overlap to 10% for a more efficient pipeline, or increase to 40% for more complex documents.

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. **Table Structure** is enabled by default.

OCR enables or disabled OCR processing when extracting text from images and scanned documents. OCR is disabled by default. This setting is best suited for processing text-based documents as quickly as possible with Docling's `DocumentConverter`. Images are ignored and not processed.

Enable OCR when you are processing documents containing images with text that requires extraction, or for scanned documents. 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 adds image descriptions generated by the SmolVLM-256M-Instruct model to OCR processing. Enabling picture descriptions can slow ingestion performance.

Knowledge ingestion flows

Flows in Langflow are functional representations of application workflows, with multiple component nodes connected as single steps in a workflow.

The **OpenSearch Ingestion** flow is the default knowledge ingestion flow in OpenRAG: when you **Add Knowledge** in OpenRAG, you run the OpenSearch Ingestion flow in the background. The flow ingests documents using **Docling Serve** to import and process documents.

This flow contains ten components connected together to process and store documents in your knowledge base.

- The **Docling Serve** component processes input documents by connecting to your instance of Docling Serve.
- The **Export DoclingDocument** component exports the processed DoclingDocument to markdown format with image export mode set to placeholder. This conversion makes the structured document data into a standardized format for further processing.

- Three **DataFrame Operations** components sequentially add metadata columns to the document data of `filename`, `file_size`, and `mimetype`.
- The **Split Text** component splits the processed text into chunks with a chunk size of 1000 characters and an overlap of 200 characters.
- Four **Secret Input** components provide secure access to configuration variables: `CONNECTOR_TYPE`, `OWNER`, `OWNER_EMAIL`, and `OWNER_NAME`. These are runtime variables populated from OAuth login.
- The **Create Data** component combines the secret inputs into a structured data object that will be associated with the document embeddings.
- The **Embedding Model** component generates vector embeddings using OpenAI's `text-embedding-3-small` model. The embedding model is selected at [Application onboarding] and cannot be changed.
- The **OpenSearch** component stores the processed documents and their embeddings in the `documents` index at `https://opensearch:9200`. By default, the component is authenticated with a JWT token, but you can also select `basic` auth mode, and enter your OpenSearch admin username and password.

All flows included with OpenRAG are designed to be modular, performant, and provider-agnostic. To modify a flow, click **Settings**, and click **Edit in Langflow**. OpenRAG's visual editor is based on the Langflow visual editor, so you can edit your flows to match your specific use case.

OpenSearch URL Ingestion flow

An additional knowledge ingestion flow is included in OpenRAG, where it is used as an MCP tool by the **Open Search Agent flow**. The agent calls this component to fetch web content, and the results are ingested into OpenSearch.

For more on using MCP clients in Langflow, see MCP clients.
To connect additional MCP servers to the MCP client, see Connect to MCP servers from your application.

Use OpenRAG default ingestion instead of Docling serve

If you want to use OpenRAG's built-in pipeline instead of Docling serve, set `DISABLE_INGEST_WITH_LANGFLOW=true` in Environment variables.

The built-in pipeline still uses the Docling processor, but uses it directly without the Docling Serve API.

For more information, see `processors.py` in the OpenRAG repository.

Environment variables

OpenRAG recognizes environment variables from the following sources:

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

Configure environment variables

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

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

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

Environment variables always take precedence over other variables.

Set environment variables

To set environment variables, do the following.

1. Stop OpenRAG.
2. Set the values in the `.env` file:

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

3. Start OpenRAG.

Updating provider API keys or provider endpoints in the `.env` file will not take effect after Application onboarding. To change these values, you must:

1. Stop OpenRAG.
2. Remove the containers:

```
docker-compose down
```

3. Update the values in your `.env` file.
4. Start OpenRAG containers.

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

5. Complete Application onboarding again.

Supported environment variables

All OpenRAG configuration can be controlled through environment variables.

AI provider settings

Configure which AI models and providers OpenRAG uses for language processing and embeddings. For more information, see Application onboarding.

Variable	Default	Description
`EMBEDDING_MODEL`	`text-embedding-3-small`	Embedding model for vector search.
`LLM_MODEL`	`gpt-4o-mini`	Language model for the chat agent.
`MODEL_PROVIDER`	`openai`	Model provider, such as OpenAI or IBM watsonx.ai.
`OPENAI_API_KEY`	-	Your OpenAI API key. Required.
`PROVIDER_API_KEY`	-	API key for the model provider.
`PROVIDER_ENDPOINT`	-	Custom provider endpoint. Only used for IBM or Ollama providers.
`PROVIDER_PROJECT_ID`	-	Project ID for providers. Only required for the IBM watsonx.ai provider.

Document processing

Control how OpenRAG processes and ingests documents into your knowledge base. For more information, see Ingestion.

Variable	Default	Description
`CHUNK_OVERLAP`	`200`	Overlap between chunks.
`CHUNK_SIZE`	`1000`	Text chunk size for document processing.
`DISABLE_INGEST_WITH_LANGFLOW`	`false`	Disable Langflow ingestion pipeline.
`DOCLING_OCR_ENGINE`	-	OCR engine for document processing.
`OCR_ENABLED`	`false`	Enable OCR for image processing.
`OPENRAG_DOCUMENTS_PATHS`	`./documents`	Document paths for ingestion.
`PICTURE_DESCRIPTIONS_ENABLED`	`false`	Enable picture descriptions.

Langflow settings

Configure Langflow authentication.

Variable	Default	Description
`LANGFLOW_AUTO_LOGIN`	`False`	Enable auto-login for Langflow.
`LANGFLOW_CHAT_FLOW_ID`	pre-filled	This value is pre-filled. The default value is found in .env.example.
`LANGFLOW_ENABLE_SUPERUSER_CLI`	`False`	Enable superuser CLI.

Variable	Default	Description
`LANGFLOW_INGEST_FLOW_ID`	pre-filled	This value is pre-filled. The default value is found in .env.example.
`LANGFLOW_KEY`	auto-generated	Explicit Langflow API key.
`LANGFLOW_NEW_USER_IS_ACTIVE`	`False`	New users are active by default.
`LANGFLOW_PUBLIC_URL`	`http://localhost:7860`	Public URL for Langflow.
`LANGFLOW_SECRET_KEY`	-	Secret key for Langflow internal operations.
`LANGFLOW_SUPERUSER`	-	Langflow admin username. Required.
`LANGFLOW_SUPERUSER_PASSWORD`	-	Langflow admin password. Required.
`LANGFLOW_URL`	`http://localhost:7860`	Langflow URL.
`NUDGES_FLOW_ID`	pre-filled	This value is pre-filled. The default value is found in .env.example.

Variable	Default	Description
`SYSTEM_PROMPT`	"You are a helpful AI assistant with access to a knowledge base. Answer questions based on the provided context."	System prompt for the Langflow agent.

OAuth provider settings

Configure OAuth providers and external service integrations.

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

OpenSearch settings

Configure OpenSearch database authentication.

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

System settings

Configure general system components, session management, and logging.

Variable	Default	Description
`LANGFLOW_KEY_RETRIES`	`15`	Number of retries for Langflow key generation.
`LANGFLOW_KEY_RETRY_DELAY`	`2.0`	Delay between retries in seconds.
`LANGFLOW_VERSION`	`latest`	Langflow Docker image version.
`LOG_FORMAT`	-	Log format (set to "json" for JSON output).
`LOG_LEVEL`	`INFO`	Logging level (DEBUG, INFO, WARNING, ERROR).
`MAX_WORKERS`	-	Maximum number of workers for document processing.
`OPENRAG_VERSION`	`latest`	OpenRAG Docker image version.
`SERVICE_NAME`	`openrag`	Service name for logging.
`SESSION_SECRET`	auto-generated	Session management.

Langflow runtime overrides

Langflow runtime overrides allow you to modify component settings at runtime without changing the base configuration.

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

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

Default values and fallbacks

When no environment variables or configuration file values are provided, OpenRAG uses default values. These values can be found in the code base at the following locations.

OpenRAG configuration defaults

These values are defined in `config_manager.py` in the OpenRAG repository.

System configuration defaults

These fallback values are defined in `settings.py` in the OpenRAG repository.

Troubleshooting

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

OpenSearch fails to start

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

Langflow connection issues

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

Memory errors

Container out of memory errors

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

Podman on macOS memory issues

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

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

Port conflicts

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

OCR ingestion fails (easyocr not installed)

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

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

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

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

1. To clear the uv cache, run:

   ```
   uv cache clean
   ```

2. Start OpenRAG:

   ```
   uvx openrag
   ```

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

Langflow container already exists

If you are running other versions of Langflow containers on your machine, you may encounter an issue where Docker or Podman thinks Langflow is already up.

Remove just the problem container, or clean up all containers and start fresh.

To reset your local containers and pull new images, do the following:

1. Stop your containers and completely remove them.

   For Podman:

   ```
   # Stop all running containers
   podman stop --all
   ```

```
# Remove all containers (including stopped ones)
podman rm --all --force
# Remove all images
podman rmi --all --force
# Remove all volumes
podman volume prune --force
# Remove all networks (except default)
podman network prune --force
# Clean up any leftover data
podman system prune --all --force --volumes
```

For Docker:

```
# Stop all running containers
docker stop $(docker ps -q)
# Remove all containers (including stopped ones)
docker rm --force $(docker ps -aq)
# Remove all images
docker rmi --force $(docker images -q)
# Remove all volumes
docker volume prune --force
# Remove all networks (except default)
docker network prune --force
# Clean up any leftover data
docker system prune --all --force --volumes
```

2. Restart OpenRAG and upgrade to get the latest images for your containers.

```
uv sync
uv run openrag
```

3. In the OpenRAG TUI, click **Status**, and then click **Upgrade**. When the **Close** button is active, the upgrade is complete. Close the window and open the OpenRAG appplication.