Lab 07

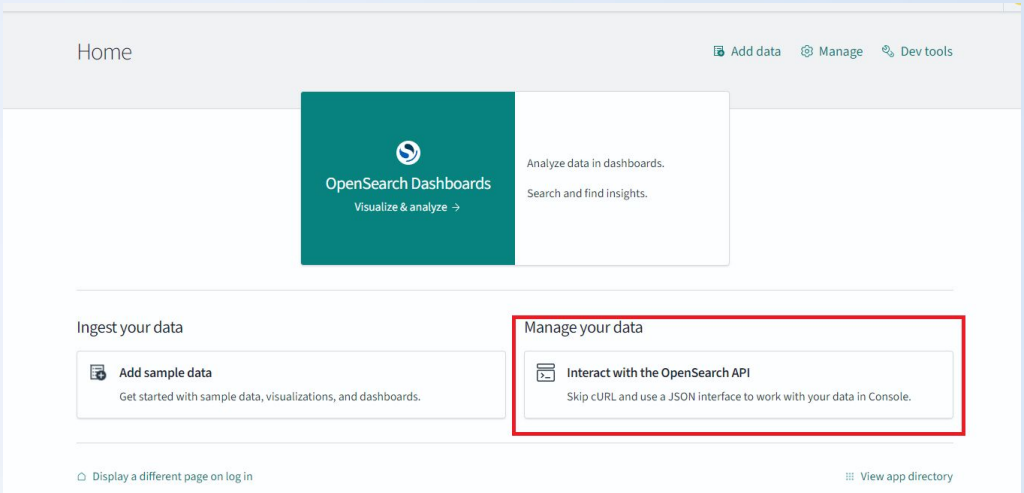
Creating Your First Agent and Model in OpenSearch

Lab Objectives

- Add a connector to our externally hosted model
- Configure the model in a search pipeline
- Create request to test inference against our pipeline and model
- Review all the changes

Log into OpenSearch

- Log into OpenSearch with the credentials that were distributed to you in class
  - https://20.106.177.39.c.hossted.app/
- While on the homepage navigate to the API servers which will be located under the developer tools



Add a connector

```
POST /_plugins/_ml/connectors/_create
{
  "name": "LMStudio Connector - $USECASE",
  "description": "Production LMStudio Host",
  "version": "2",
  "protocol": "http",
  "parameters": {
    "endpoint": "http://20.106.179.227:5000",
    "model": "openvoid/prox-7b-dpo-gguf",
    "max tokens": 2000,
    "temperature": 0.5
  },
  "credential": {
    "openAI_key": "fakeapikey"
    },
  "client_config": {
    "connection_timeout": 1600,
    "read_timeout": 1800,
    "retry_backoff_millis": 1800000,
    "retry_timeout_seconds": 1800
  },
  "actions": [
    {
      "action_type": "predict",
      "method": "POST",
      "url": "${parameters.endpoint}/v1/chat/completions",
      "request_body": """{ "model": "${parameters.model}", "messages":
${parameters.messages}, "temperature": ${parameters.temperature} }"""
    }
  ]
}
```

POST /_plugins/_ml/connectors/_create: Creates a new connector

- **name**: The name of the connector (`"LMStudio Connector - $USECASE"`)
- **description**: A brief description of the connector (`"Production LMStudio Host"`)
- **version**: Version number of the connector (`"2"`)
- **protocol**: Communication protocol used (`"http"`)
- **parameters**: Contains the configuration parameters for the connector
  - **endpoint**: The API endpoint URL (`"http://20.106.179.227:5000"`)
  - **model**: The model to be used (`"openvoid/prox-7b-dpo-gguf"`)
  - **temperature**: The temperature setting for the model (`0.5`)
- **actions**: Defines the actions the connector can perform
  - **action_type**: The type of action (`"predict"`)
  - **method**: HTTP method to be used (`"POST"`)
  - **url**: URL template for making predictions (`"${parameters.endpoint}/v1/chat/completions"`)

Add a connector(continued)

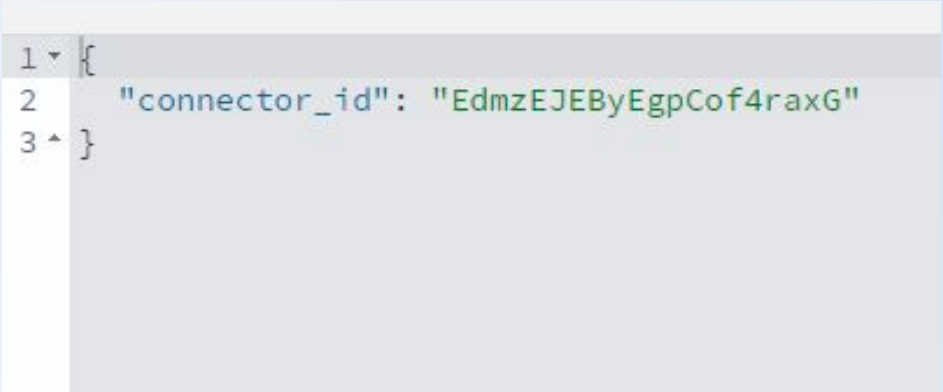
```
POST /_plugins/_ml/connectors/_create
{
  "name": "LMStudio Connector - $USECASE",
  "description": "Production LMStudio Host",
  "version": "2",
  "protocol": "http",
  "parameters": {
    "endpoint": "http://20.106.179.227:5000",
    "model": "openvoid/prox-7b-dpo-gguf",
    "max tokens": 2000,
    "temperature": 0.5
  },
  "credential": {
    "openAI_key": "fakeapikey"
    },
  "client_config": {
    "connection_timeout": 1600,
    "read_timeout": 1800,
    "retry_backoff_millis": 1800000,
    "retry_timeout_seconds": 1800
  },
  "actions": [
    {
      "action_type": "predict",
      "method": "POST",
      "url": "${parameters.endpoint}/v1/chat/completions",
      "request_body": """{ "model": "${parameters.model}", "messages":
${parameters.messages}, "temperature": ${parameters.temperature} }"""
    }
  ]
}
```

- **credential**: API Keys that may be needed
  - openAI_key: API Key for OpenAPI
  - LMStudio does not need one, but Opensearch requires one to be input- we are able to put gibberish here.
- **Client_config**: Configuring client-side parameters
  - `connection_timeout`: Max wait time to establish a connection (ms).
  - `read_timeout`: Max wait to receive data (s).
  - `retry_backoff_millis`: Time between retries (ms).
  - `retry_timeout_seconds`: Total retry period allowed (s).
- **actions**: Defines the actions the connector can perform
  - **action_type**: The type of action (`"predict"`)
  - **method**: HTTP method to be used (`"POST"`)
  - **url**: URL template for making predictions (`"${parameters.endpoint}/v1/chat/completions"`)

Add a connector

- When you submit the command to create the connector you will receive a **connector id**

- Notate/write down this **connector id** for the next steps
- An example of what this may look like
    - Note: this is just an example! Please use your own connector id in the next steps



Register model and deploy

- POST /_plugins/_ml/models/_register
- {
- "name": "lab7-rag-model",
- "function_name": "remote",
- "description": "OpenVOID model demo",
- "connector_id": "**EdmzEJEByEgpCof4raxG**"
- }

- POST /_plugins/_ml/models/**EdmzEJEByEgpCof4raxG**/_deploy

Registering a Model

- **POST /_plugins/_ml/models/_register**
  - **name**: The name of the model (`"lab7-rag-model"`)
  - **function_name**: The function or purpose of the model (`"remote"`)
  - **description**: A brief description of the model (`"Mistral model demo"`)
  - **connector_id**: The ID of the connector used (`"EdmzEJEByEgpCof4raxG"`)

Deploying a Model

  - Deploys the specified model
  - The model ID (`Mtm6EJEByEgpCof4HqwJ`) should be replaced with the actual model ID you want to deploy

Register and deploy

- When you submit the command to create the model group you will receive a **model_group_id**
- Notate/write down this **model_group_id** for the next steps
- When you submit the command to register the model you will receive a **model_id**
- Notate/write down this **model_id** for the next step


```
{
  "model_group_id": "Ldm5EJEByEgpCof43qyF",
  "status": "CREATED"
}
```

```
{
  "task_id": "Mdm6EJEByEgpCof4HazZ",
  "status": "CREATED",
  "model_id": "Mtm6EJEByEgpCof4HqwJ"
}
```

```
{
  "task_id": "S9nBEJEByEgpCof4Uqwq",
  "task_type": "DEPLOY_MODEL",
  "status": "COMPLETED"
}
```

Use the model in a pipeline

```
PUT /_search/pipeline/openai_pipeline
{
 "response_processors": [
        {
        "retrieval_augmented_generation": {
        "tag": "lab7",
        "description": "lab7-rag-pipeline",
        "model_id": "UcPiHZABM8E_-Dt1oSWN",
        "context_field_list": ["log", "message"],
        "system_prompt": "You are a helpful assistant. Never make up an answer. If you're not sure of something, you MUST say that you don't know.",
        "user_instructions": "I will send you some information that MAY be relevant to the context of the question."
        }
        }
 ]
}
```

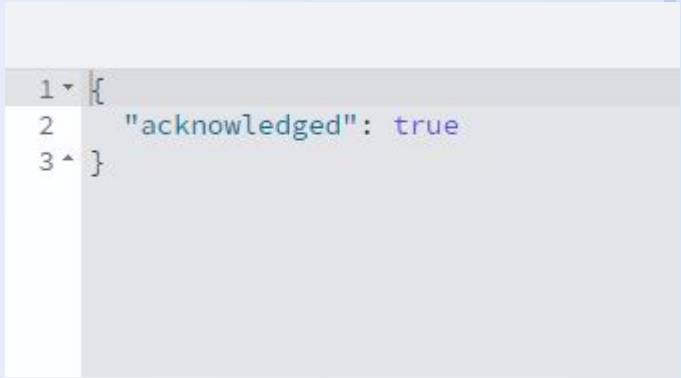
retrieval_augmented_generation: Specifies the RAG processor

- **tag**: A label for the processor (`"lab7"`)
- **description**: Description of the pipeline (`"lab7-rag-pipeline"`)
- **model_id**: Identifier for the model used (`"UcPiHZABM8E_-Dt1oSWN"`)
- **context_field_list**: Fields to include in the context (`["severity", "message"]`)
- **system_prompt**: Instructions for the AI model:
  - `"You are a helpful assistant. Never make up an answer. If you're not sure of something, you MUST say that you don't know."`
- **user_instructions**: Instructions provided to users

Use the model in a pipeline

- Note that you will only see **acknowledged: true** after the previous step
  - This does not mean it was successful but rather it was fully qualified syntax

- A search response processor intercepts a search response and search request (the query, results, and metadata passed in the request), performs an operation with or on the search response, and returns the search response



Making a request

- This is the query we use to send our prompt + RAG parameters to the model
- This is a large query so you may copy this from the OpenSearch lab commands text document from Day2 in the git repository
- The next page contains a description on this query

```
GET /weblogs_raw/_search?search_pipeline=openai_pipeline
{
 "query": {
  "bool": {
   "filter": [
    {
     "exists": {
      "field": "log"
     }
    }
   ],
   "must": [
    {
     "match": {
      "message": {
       "query": "which modules were loaded?"
      }
     }
    }
   ]
  }
 },
 "ext": {
  "generative_qa_parameters": {
   "llm_model": "openvoid/prox-7b-dpo-gguf",
   "llm_question": "which modules were loaded?",
   "context_size": 20,
   "timeout": 30
  }
 }
}
```

Making a Request

- **Query**: The main search query object
  - **bool**: Combines multiple query clauses
    - **filter**: Restricts results to documents that meet certain criteria
      - **exists**: Checks if a field is present
        - **field**: The field that must exist (`severity`)
    - **must**: Specifies mandatory conditions for document matching
      - **match**: Searches for documents where the field contains the specified value
        - **message**: The field to search
          - **query**: The search phrase (`"which modules were loaded?"`)
- **ext**: Contains extensions for additional processing
  - **generative_qa_parameters**: Parameters for generative question answering
    - **llm_model**: The language model to use (`"openvoid/prox-7b-dpo-gguf"`)
    - **llm_question**: The question to be answered (`"which modules were loaded?"`)
    - **context_size**: Size of the context window (20)
    - **timeout**: Time limit for the query (30 seconds)

Register and deploy

200 - OK 2878 ms


```
{
  "took": 1,
  "timed_out": false,
  "_shards": {
    "total": 1,
    "successful": 1,
    "skipped": 0,
    "failed": 0
  },
  "hits": {
    "total": {
      "value": 0,
      "relation": "eq"
    },
    "max_score": null,
    "hits": []
  },
  "ext": {
    "retrieval_augmented_generation": {
      "answer": " The Linux kernel maintains a list of all the modules that have been loaded into memory. This information can be accessed using the lsmod command. The output of this command will include details such as the name of each module, its size in kilobytes, and the number of times it has been used. If you want to know which modules were loaded, running the lsmod command is the way to go."
    }
  }
}
```

Lab End