RAG Integration to Aggrag

Adding a new RAG to the ragstore in Aggrag

This documentation outlines two methods for integrating a new RAG (Retrieval-Augmented Generation) component into the Aggrag ragstore, directory where all rags are stored. Each method has its own approach and requirements, allowing users to choose the one that best fits their needs.

Method 1: Integration with Aggrag

This method involves adding a new RAG directly to Aggrag codebase. The following two components constitute Method 1:

1. Adding a New RAG: Backend (BE)

This includes defining the new RAG, implementing required methods, updating the `RagStoreBool` class, modifying the `create_ragstore` method, and creating new RAG settings.

2. Adding a New RAG: Frontend (FE)

This involves ensuring the new RAG component is defined in the backend, updating the `ModelSettingSchemas.tsx`, modifying the `store.tsx`, and updating the `models.ts` file to include the new RAG provider.

Step 1: Define the new RAG:

Create a new class for the RAG. This class should implement the necessary methods for processing queries, creating indexes, and retrieving data.

Ensure that the new RAG component follows the same interface of Aggrag architecture as existing components are doing (e.g., `Base`, `SubQA`, `Raptor`, `MetaLlama`, `MetaLang`).

Ensure that the new RAG component implements all required methods such as `create_index_async`, `retrieve_index_async`, `documents_loader` specific to new RAG and other chat-related methods (e.g., `achat`).

Step 2: Update the `RagStoreBool` Class:

Modify the `RagStoreBool` class in “chainforge/aggrag/core/schema.py” to include a boolean attribute for the new RAG component. This will allow you to enable or disable the component when initializing the `AggRAG` class.

```
   class RagStoreBool:
       def __init__(self, base=False, subqa=False, raptor=False, meta_llama=F
alse, meta_lang=False, new_rag=False):
           self.base = base
           self.subqa = subqa
           self.raptor = raptor
           self.meta_llama = meta_llama
           self.meta_lang = meta_lang
           self.new_rag = new_rag  # Add new RAG component
```

Step 3: Modify the `create_ragstore` Method:

In the `create_ragstore` method of the `AggRAG` class, add logic to initialize the new RAG component based on the boolean flag in `RagStoreBool`.

```
   def create_ragstore(ragstore_bool: RagStoreBool) -> RagStore:
       return RagStore(
           ...
           new_rag=NewRAG(
               usecase_name=self.usecase_name,
               iteration=self.iteration,
               DATA_DIR=self.DATA_DIR,
               new_rag_setting=self.ragstore_settings.new_rag_setting if self
.ragstore_settings.new_rag_setting else NewRagSetting()
           ) if ragstore_bool.new_rag else None,
       )
```

To provide new RAG configurable parameters, create a settings class (e.g., `NewRagSetting` , a pydantic setting class) in chainforge/aggrag/core/schema.py to hold configuration parameters with default values for the new RAG.

```
from pydantic import BaseModel, Field
from typing import Literal

class NewRagSetting(BaseModel):
    chunk_size: int = 512
    system_prompt: str = DEFAULT_SYSTEM_PROMPT
    context_prompt: str = DEFAULT_CONTEXT_PROMPT
    temperature: float = 0.1
    index_name: Literal['new_rag_index'] = "new_rag_index"  # Unique index
name for the new RAG
```

Ensure that the new RAG component is defined in the backend, similar to how `MetaLlama` and `MetaLang` are defined in the `aggrag.py` and `schema.py` files. This includes creating a settings class and ensuring it is included in the `RagStoreSettings`.

Step 5: Update ModelSettingSchemas.tsx:

Create a new settings schema for the new RAG component in `ModelSettingSchemas.tsx`. This schema should define the properties and types for the new RAG, similar to how `Meta_llamaRagSettings` and `Meta_langRagSettings` are structured.

```
const NewRagSettings: ModelSettingsDict = {
  fullName: "NewRag",
  schema: {
    type: "object",
    required: ["shortname"],
    properties: {
      shortname: {
        type: "string",
        title: "Nickname",
        description: "Unique identifier to appear in UI. Keep it short.",
        default: "new_rag",
      },
      ai_service: {
        default: "AzureOpenAI",
        oneOf: [
          { const: "OpenAI", title: "OpenAI" },
          { const: "AzureOpenAI", title: "AzureOpenAI" },
          { const: "Nvidia", title: "Nvidia" },
        ],
        title: "AI service",
        type: "string",
      },
      // Add other necessary properties here
    },
  },
  uiSchema: {
    "ui:submitButtonOptions": {
      props: {
        disabled: false,
        className: "mantine-UnstyledButton-root mantine-Button-root",
      },
      norender: false,
      submitText: "Submit",
    },
    shortname: {
      "ui:autofocus": true,
    },
    // Add UI schema for other properties if needed
  },
```

```
  postprocessors: {},
};
```

Add the new RAG settings to the `ModelSettings` dictionary.

```
export const ModelSettings: Dict<ModelSettingsDict> = {
  "gpt-3.5-turbo": ChatGPTSettings,
  "gpt-4": GPT4Settings,
  // ... other existing models ...
  base: BaseRagSettings,
  subqa: SubQARagSettings,
  raptor: RaptorRagSettings,
  meta_llama: Meta_llamaRagSettings,
  meta_lang: Meta_langRagSettings,
  new_rag: NewRagSettings,  // Add new RAG settings
};
```

Update the Function to Get RAG Settings Schema:

Modify the `getSettingsSchemaForRAG` function to include the new RAG.

```
export function getSettingsSchemaForRAG(
  rag_name: string,
): ModelSettingsDict | undefined {
  const rag_provider = getRAGProvider(rag_name);

  const provider_to_settings_schema: {
    [K in RAGProvider]?: ModelSettingsDict;
  } = {
    [RAGProvider.Base]: BaseRagSettings,
    [RAGProvider.SubQA]: SubQARagSettings,
    [RAGProvider.Raptor]: RaptorRagSettings,
    [RAGProvider.Meta_llama]: Meta_llamaRagSettings,
    [RAGProvider.Meta_lang]: Meta_langRagSettings,
    [RAGProvider.NewRag]: NewRagSettings,  // Add new RAG provider
  };

  if (rag_provider === RAGProvider.Custom) return ModelSettings[rag_name];
  else if (rag_provider && rag_provider in provider_to_settings_schema)
    return provider_to_settings_schema[rag_provider];
  else {
    console.error(`Could not find provider for rag ${rag_name}`);
    return undefined;
  }
}
```

Step 6: Update store.tsx:

Add the new RAG component to the `initRAGProviderMenu` in `store.tsx`. This will allow users to select the new RAG from the UI.

```
export const initRAGProviderMenu: LLMSpec[] = [
  {
    name: "NewRag",
    emoji: "✨",
    model: "new_rag",
    base_model: "new_rag",
    temp: 0.7,
    description: "New RAG component for enhanced capabilities.",
  },
];
```

Step 7: Update the `models.ts` file:

Add the New RAG Provider to the Enum:

Update the `NativeRAG` enum to include the new RAG.

```
export enum NativeRAG {
  Base = "base",
  SubQA = "subqa",
  Raptor = "raptor",
  Meta_llama = "meta_llama",
  Meta_lang = "meta_lang",
  NewRag = "new_rag",  // Add new RAG provider
}
```

Add the new RAG provider to the `RAGProvider` enum in `models.ts`.

```
export enum RAGProvider {
  Base = "base",
  SubQA = "subqa",
  Raptor = "raptor",
  Meta_llama = "meta_llama",
  Meta_lang = "meta_lang",
  NewRag = "new_rag",  // Add new RAG provider
  Custom = "__custom",
}
```

Update the `getRAGProvider` function to recognize the new RAG.

```
export function getRAGProvider(rag: LLM): RAGProvider | undefined {
  const rag_name = getEnumName(NativeRAG, rag.toString());
  if (rag_name?.startsWith("Base")) return RAGProvider.Base;
  else if (rag_name?.startsWith("SubQA")) return RAGProvider.SubQA;
  else if (rag_name?.startsWith("Raptor")) return RAGProvider.Raptor;
  else if (rag_name?.startsWith("Meta_llama")) return RAGProvider.Meta_llama;
  else if (rag_name?.startsWith("Meta_lang")) return RAGProvider.Meta_lang;
  else if (rag_name?.startsWith("NewRag")) return RAGProvider.NewRag;  // Add
 new RAG check
  else if (rag.toString().startsWith("__custom/")) return RAGProvider.Custom;
```

```
  return undefined;
}
```

Step 8: Changes to `utils.ts`:

Update the `index_file` Function:

Add handling for the new RAG in the `index_file` function.

```
export async function index_file(params: Dict): Promise<Dict> {
  const query: Dict = {
    files_path: params.files_path,
    rag_name: params.rag_name,
    ragstore_settings: {},
  };
  const rag_provider: RAGProvider | undefined = getRAGProvider(
    params.rag_name as RAG,
  );
  if (rag_provider === RAGProvider.Base)
    query.ragstore_settings.base_rag_setting = params.settings || {};
  else if (rag_provider === RAGProvider.Raptor)
    query.ragstore_settings.raptor_rag_setting = params.settings || {};
  else if (rag_provider === RAGProvider.SubQA)
    query.ragstore_settings.subqa_rag_setting = params.settings || {};
  else if (rag_provider === RAGProvider.Meta_lang)
    query.ragstore_settings.meta_lang_rag_setting = params.settings || {};
  else if (rag_provider === RAGProvider.Meta_llama)
    query.ragstore_settings.meta_llama_rag_setting = params.settings || {};
+  else if (rag_provider === RAGProvider.NewRag)  // Add new RAG handling
+    query.ragstore_settings.new_rag_setting = params.settings || {};

  const { response, error } = await call_flask_backend("indexRAGFiles", query
);

  // Fail if an error is encountered
  if (error !== undefined || response === undefined) throw new Error(error);

  return response;
}
```

Update the `rag_store_chat` Function:

Add handling for the new RAG in the `rag_store_chat` function.

```
export async function rag_store_chat(
  prompt: string,
  model: LLM,
  params: Dict,
  should_cancel?: () => boolean,
): Promise<[Dict, Dict]> {
  const query: Dict = {
    rag_name: model,
```

```
    query: prompt,
    index_path: params.index_path,
    ragstore_settings: {},
  };
  const rag_provider: RAGProvider | undefined = getRAGProvider(model as RAG);
  if (rag_provider === RAGProvider.Base)
    query.ragstore_settings.base_rag_setting = params.settings || {};
  else if (rag_provider === RAGProvider.NewRag)  // Add new RAG handling
     query.ragstore_settings.new_rag_setting = params.settings || {};

  let response: Array<Dict> = [];
  // Abort if canceled
  if (should_cancel && should_cancel()) throw new UserForcedPrematureExit();

  try {
    const temp_response = await call_flask_backend("RAGStoreChat", query);
    if (temp_response?.response?.length > 0) response = temp_response.respons
e;
    else throw new Error(temp_response?.error);
  } catch (error: any) {
    if (error?.response) {
      throw new Error(error.response.data?.error?.message);
    } else {
      console.log(error?.message || error);
      throw new Error(error?.message || error);
    }
  }
  return [query, response];
}
```

Step 9: Changes to `flask_app.py`:

Update the `indexRAGFiles()` Function:

```
rag_models = ["base", "raptor", "subqa", "meta_llama", "meta_lang",
"new_rag"]


    allowed_rag_settings = {
        "base_rag_setting",
        "subqa_rag_setting",
        "raptor_rag_setting",
        "meta_llama_rag_setting",
        "meta_lang_rag_setting",
        "new_rag_setting",
    }
    if not set(ragstore_settings.keys()).issubset(allowed_rag_settings):
        return jsonify(
            {
                "error": 'ragstore_settings should be one of
["base_rag_setting", "subqa_rag_setting", "raptor_rag_setting",
"meta_llama_rag_setting", "meta_lang_rag_setting", "new_rag_setting"]'
            }
```

```
        )

    try:
        aggrag_ragstore_settings = RagStoreSettings(**ragstore_settings)
        ragstore_settings = RagStoreSettings(
            base_rag_setting=(
                BaseRagSetting(**ragstore_settings.get("base_rag_setting"))
                if ragstore_settings.get("base_rag_setting")
                else None
            ),

            raptor_rag_setting=(

               RaptorRagSetting(**ragstore_settings.get("raptor_rag_setting")
               )
                if ragstore_settings.get("raptor_rag_setting")
                else None
            ),
            subqa_rag_setting=(
                SubQARagSetting(**ragstore_settings.get("subqa_rag_setting"))
                if ragstore_settings.get("subqa_rag_setting")
                else None
            ),
            meta_llama_rag_setting=(

               MetaLlamaRagSetting(**ragstore_settings.get("meta_llama_rag_se
               tting"))
                if ragstore_settings.get("meta_llama_rag_setting")
                else None
            ),
            meta_lang_rag_setting=(

               MetaLangRagSetting(**ragstore_settings.get("meta_lang_rag_sett
               ing"))
                if ragstore_settings.get("meta_lang_rag_setting")
                else None
            ),
            new_rag_setting=(
                NewRagSetting(**ragstore_settings.get("new_rag_setting"))
                if ragstore_settings.get("new_rag_setting")
                else None
            ),
        )
```

Update the `RAGStoreChat()` Function:

```
rag_models = ["base", "raptor", "subqa", "meta_llama", "meta_lang",
"new_rag"]

    allowed_rag_settings = {
        "base_rag_setting",
        "subqa_rag_setting",
        "raptor_rag_setting",
```

```
        "meta_llama_rag_setting",
        "meta_lang_rag_setting",
        "new_rag_setting",
    }
    if not set(ragstore_settings.keys()).issubset(allowed_rag_settings):
        return jsonify(
            {
                "error": 'ragstore_settings should be one of
["base_rag_setting", "subqa_rag_setting", "raptor_rag_setting",
"meta_llama_rag_setting", "meta_lang_rag_setting", "new_rag_setting]'
            }
        )

    try:
        aggrag_ragstore_settings = RagStoreSettings(**ragstore_settings)
        ragstore_settings = RagStoreSettings(
            base_rag_setting=(
                BaseRagSetting(**ragstore_settings.get("base_rag_setting"))
                if ragstore_settings.get("base_rag_setting")
                else None
            ),

            raptor_rag_setting=(
              RaptorRagSetting(**ragstore_settings.get("raptor_rag_setting"))
                if ragstore_settings.get("raptor_rag_setting")
                else None
            ),
            subqa_rag_setting=(
                SubQARagSetting(**ragstore_settings.get("subqa_rag_setting"))
                if ragstore_settings.get("subqa_rag_setting")
                else None
            ),
            meta_llama_rag_setting=(
MetaLlamaRagSetting(**ragstore_settings.get("meta_llama_rag_setting"))
                if ragstore_settings.get("meta_llama_rag_setting")
                else None
            ),
            meta_lang_rag_setting=(
MetaLangRagSetting(**ragstore_settings.get("meta_lang_rag_setting"))
                if ragstore_settings.get("meta_lang_rag_setting")
                else None
            ),
            new_rag_setting=(
            NewRagSetting(**ragstore_settings.get("new_rag_setting"))
                if ragstore_settings.get("new_rag_setting")
                else None
            ),
        )
```

Conclusion :

By following these steps, you can effectively add a new RAG component to the `AggRAG` class. Ensure that you maintain consistency with existing components and thoroughly test the new functionality to ensure it integrates seamlessly with the rest of the application.

How to Add a New AI Service and Its LLMs to AggRAG

Step 1: Test AI Service and Models with Llama-Index

Before integrating a new AI service into AggRAG, verify that the service and its models are supported by `llama-index` and function correctly. Set up a Jupyter notebook environment to test the same.

Step 2: Identify the Configuration file:

Locate the "CommonRagSettings" object in "aggrag\library\react-server\src\ModelSettingSchemas.tsx" file, which defines AI services and their models.

Step 3: Add the New AI Service to Configuration:

Navigate to `CommonRagSettings`: This object contains `if` conditions specifying configurations for different AI services.

Append a New Condition: Add a new object to the `allOf` array within `CommonRagSettings` for the new AI service and its models.

Step 4: Define the AI Service and Its Models:

Specify the AI Service Name: Create an `if` condition within the new object for the AI service name.

```
if: {
  properties: {
    ai_service: {
      const: "YourNewAIService",
    },
  },
},
```

Define Available Models**:** In the `then` section, specify `llm_model` properties, including a default model and an enumeration of all models.

```
then: {
  properties: {
    llm_model: {
      default: "default-model-name",
      enum: ["model-name-1", "model-name-2"],
      title: "LLM Model",
      type: "string",
    },
  },
},
```

Step 5: Update the .env Configuration:

Add the new AI service and its models to the `AI_SERVICES_CONFIG` JSON structure in the `.env` file. This step ensures the service is recognized and its models are available for use.

Open the `.env` File: Locate the `AI_SERVICES_CONFIG` variable.

Insert the AI Service Configuration: Add a JSON snippet for the new service and its models.

```
"YourNewAIService": {
    "chat_models": {
        "model-name-1": {
            "model_name": "model-name-1"
        },
        "model-name-2": {
            "model_name": "model-name-2"
        }
        // Add additional models as needed
    }
}
```

Ensure JSON Validity: Correctly place commas and maintain proper JSON structure.

Step 6: Update the Documentation:

Reflect these changes in the project documentation, detailing the new AI service, its models, and any specific usage instructions.

Step 7: Test the Integration:

Select the new AI service through the UI, verify model availability, and check for errors or issues.

Conclusion:

Following these steps allows developers to extend AggRAG's capabilities by integrating additional AI services and models, enhancing the platform's versatility.

Method 2: Independent RAG Integration

This approach is independent of the Aggrag codebase and requires only a runnable RAG Python script along with a decorator from Chainforge in its response rendering method. Although this documentation includes the Aggrag approach to build the runnable RAG Python script, as it simplifies the integration process for new users to aggrag later. Users can choose between

developing a custom provider script using the Aggrag approach or opting for a simpler implementation.

Step 1: Create a Python Script for the New RAG

Create a new `.py` file that will serve as the script for your new RAG. This script should include all crucial components such as document loading, retrieval, and response rendering.

Example:

```
# my_new_rag.py

class MyNewRAG:
    def __init__(self, data_dir):
        self.data_dir = data_dir
        # Initialize other components as needed

    def load_documents(self):
        # Logic to load documents
        pass

    def retrieve(self, query):
        # Logic to retrieve relevant documents based on the query
        pass

    def chat(self, response):
        # Logic to format and return the response
        pass
```

Step 2: Identify Configurable Components

Identify the configurable components in your RAG, such as models, temperature settings, prompts, etc. Make a comprehensive list of these items.

Example:

- **Models**: `gpt-35-turbo`, `gpt-4`
- **Temperature**: Default value of `0.1`
- **Prompts**: `system_prompt`, `context_prompt`

Step 3: Create a rag_settings_schema

Use the items collected in Step 2 to create a `RAG_SETTINGS_SCHEMA`. This schema will define the structure and constraints for the configuration settings of your RAG.

Example:

```
RAG_SETTINGS_SCHEMA = {
    "type": "object",
    "properties": {
        "temperature": {
            "type": "number",
            "default": 1.0,
            "description": "Controls the randomness of the output."
        },
        "system_prompt": {
            "type": "string",
            "description": "The system prompt for the model."
        },
        "context_prompt": {
            "type": "string",
            "description": "The context prompt for the model."
        },
        # Add other configurable components here
    },
    "ui": {
        "temperature": {
            "ui:help": "Defaults to 1.0.",
            "ui:widget": "range"
        },
        "system_prompt": {
            "ui:widget": "textarea",
            "ui:rows": 10,
            "ui:cols": 50
        },
        "context_prompt": {
            "ui:widget": "textarea",
            "ui:rows": 10,
            "ui:cols": 50
        },
    }
}
```

For more details on JSON schema forms, refer the following links.
[react-jsonschema-form playground (rjsf-team.github.io)]

Step 4: Integrate AggRAG Flow

Integrate the AggRAG flow in your script by including the `AggRAG` class code and it's other required components. Ensure that all necessary imports are resolved within the same file, as Chainforge accepts a single Python file for custom provider functionality.

Example:

```
#from aggrag.chainforge.aggrag import AggRAG
''' Note: Include the imported components code in .py file as chanifoge
 accepts single file only to work as custom provider '''
class MyNewRAG:
    # Initialization and methods as defined earlier

# Integrate AggRAG
aggrag_instance = AggRAG(ragstore_bool=RagStoreBool(base=True), ...)
```

Step 5: Decorate the Driver Method

Decorate the driver method with the `@provider` decorator imported from Chainforge. This will allow your method to be recognized as a provider within the Aggrag ecosystem.

Example:

```
from chainforge.providers import provider

@provider(name="my_new_rag", emoji=":)", settings_schema=RAG_SETTINGS_SCHEMA)
def run_my_new_rag(prompt: str, model: str, **kwargs):
    # Logic to run the RAG and return responses
    pass
```

Step 6: Reference Existing Examples

Take reference from existing examples in the `examples` folder, such as `sample_base.py` and `sample_raptor.py`. These examples can provide insights into how to structure your code and implement various functionalities.

Step 7: Test and Integrate

Once you can see responses in the UI, gradually move towards integrating your new RAG with the existing AggRAG system. Ensure that it works seamlessly with other components and that you can retrieve and process responses as expected.

Conclusion:

By following these steps, you can effectively integrate a new RAG into the Aggrag system. Ensure thorough testing at each stage to confirm that all components function correctly and that the integration meets your requirements.