The `app/backend` folder of the Azure Search OpenAI demo repository contains the backend logic for the application. Here's a brief overview of its structure and purpose:

1. **`approaches/`: Contains different search approaches implemented in the project.**
2. **`core/`: Includes core functionalities such as authentication and configuration.**
3. **`prepdocslib/`: Likely contains scripts for preparing documents for search.**
4. **`app.py`: The main application entry point.**
5. **`config.py`: Configuration settings for the application.**
6. **`decorators.py`: Custom decorators for various purposes.**
7. **`error.py`: Error handling utilities.**
8. **`gunicorn.conf.py`: Configuration for the Gunicorn server.**
9. **`main.py`: Main script to run the application.**
10. **`prepdocs.py`: Script for preparing documents.**
11. **`requirements.in` and `requirements.txt`: Dependency lists.**
12. **`text.py`: Text processing utilities.**

The core of this repository demonstrates how to integrate Azure Search with OpenAI to provide advanced search capabilities.

Approaches:

The `approaches` directory in the Azure-Samples/azure-search-openai-demo repository contains different classes that define various strategies (approaches) for integrating Azure Cognitive Search with OpenAI's language models. Each file typically represents a distinct approach to querying and combining the results from these services. Here's an explanation of each file in the directory:

1. **`approach.py`**
    - This is likely an abstract base class or an interface defining the common structure and methods that all specific approaches must implement. It might include methods for initializing the approach, querying the search service, processing results, and possibly interfacing with the OpenAI model.
2. **`chatreadretrieverecent.py`**
    - This file probably defines an approach that integrates chat functionality, reading from a source, retrieving relevant documents, and considering recent context. It might combine conversation history with recent documents or interactions to provide more accurate and contextually relevant responses.
3. **`chatreadretrieveread.py`**
    - Similar to `chatreadretrieverecent.py`, but this approach might focus on combining chat functionality with reading and retrieving documents without the specific emphasis on recent interactions. It could be aimed at fetching and reading documents based on a broader context rather than just recent history.
4. **`readdecomposeask.py`**
    - This file likely implements an approach that involves reading documents, decomposing the content into smaller, manageable parts, and then generating questions (possibly using OpenAI's models). This can be useful for creating a more interactive and

detailed understanding of the content by breaking down complex documents into simpler queries and answers.

5. `readretrieveread.py`
    - This approach might involve a cycle of reading a query, retrieving relevant documents, and then reading those documents to provide a comprehensive response. This iterative process helps in refining the search results and generating more precise answers by continuously integrating feedback from the retrieved documents.

Each of these files provides a unique method of combining Azure Cognitive Search and OpenAI's language models to handle different types of queries and data interactions. By structuring the approaches in separate files, the codebase allows for flexibility and modularity, making it easier to experiment with and implement various search and retrieval strategies.

Approaches.py

The provided code from the `approaches` directory serves as a foundational base class for different query and retrieval strategies. Here's an explanation of its key components:

`Document` Class

The `Document` class is a data model for storing and handling information about retrieved documents. It includes various attributes like `id`, `content`, `embedding`, `image_embedding`, `category`, `sourcepage`, `sourcefile`, `oids`, `groups`, `captions`, `score`, and `reranker_score`.

Methods:

- **`serialize_for_results`**: Converts the document's attributes into a dictionary suitable for API responses.
- **`trim_embedding`**: Trims the embedding list to a more readable format, showing only the first two elements and indicating the count of remaining elements.

`ThoughtStep` Class

The `ThoughtStep` class models a step in the reasoning or processing sequence, encapsulating a `title`, `description`, and optional `props`.

`Approach` Class

The `Approach` class is an abstract base class (ABC) defining a blueprint for different search and retrieval strategies. It includes several methods and attributes necessary for interacting with Azure Cognitive Search and OpenAI models.

Constructor (`__init__`)

Initializes the class with various dependencies and configuration options, including:

- `search_client`: The Azure Cognitive Search client.
- `openai_client`: The OpenAI client.
- `auth_helper`: Helper for handling authentication.
- `query_language`, `query_speller`: Language and spelling configurations for queries.
- `embedding_deployment`, `embedding_model`, `embedding_dimensions`: Details for handling embeddings.
- `openai_host`, `vision_endpoint`, `vision_token_provider`: Configuration for OpenAI and vision API interactions.

Methods:

- **`build_filter`**: Constructs a search filter based on overrides and authentication claims.
- **`search`**: Executes a search query using the Azure Search client, optionally leveraging semantic ranking and captions.
- **`get_sources_content`**: Retrieves the content from source documents, optionally using semantic captions.
- **`get_citation`**: Generates a citation string for a document, handling different types of sources (e.g., image vs. text).
- **`compute_text_embedding`**: Computes a text embedding using the OpenAI client.
- **`compute_image_embedding`**: Computes an image embedding by interacting with the vision API.
- **`run`**: Abstract method to be implemented by subclasses, defining how a query should be executed.
- **`run_stream`**: Abstract method for executing a query and returning a stream of results.

Integration

The `Approach` class sets the stage for specific implementations that would reside in files like `chatreadretrieverecent.py`, `chatreadretrieveread.py`, `readdecomposeask.py`, and

`readretrieveread.py`. Each of these files would define a subclass of `Approach`, implementing the `run` and `run_stream` methods to handle queries in different ways, combining the strengths of Azure Cognitive Search and OpenAI models.

Summary

In summary, the `approaches` directory defines a structured way to implement various strategies for querying and retrieving information using Azure Cognitive Search and OpenAI. The `Approach` class provides a comprehensive framework for these implementations, handling everything from search filtering to embedding computation and result processing.

Step by Step Breakdown of Code of Approaches.py:

Imports

1. **Standard Library Imports:**
    - `os`: Provides a way of using operating system dependent functionality.
    - `abc`: Provides tools for defining Abstract Base Classes.
    - `dataclasses`: Provides a decorator and functions for automatically adding special methods to classes, such as `__init__`.
    - `typing`: Offers a standard way to specify type hints.
    - `urllib.parse`: Provides functions for parsing URLs.
2. **Third-Party Library Imports:**
    - `aiohttp`: An asynchronous HTTP client/server framework.
    - `azure.search.documents.aio`: The Azure SDK for asynchronous search client operations.
    - `azure.search.documents.models`: Provides models for working with Azure Cognitive Search, such as query types and vector queries.
    - `openai`: The OpenAI API client.
    - `openai.types.chat`: Provides types for OpenAI's chat completions.
3. **Local Imports:**
    - `core.authentication`: Imports `AuthenticationHelper`, presumably a helper class for handling authentication.
    - `text`: Imports `nonewlines`, a utility function that likely removes new lines from a string.

Explanation of Imported Modules and Functions

Standard Library

- **`os`**:
  - Used for interacting with the operating system, such as file paths and environment variables.
- **`abc.ABC`**:
  - Provides a way to define abstract base classes. An abstract base class can have abstract methods that must be implemented by subclasses.
- **`dataclasses.dataclass`**:
  - A decorator to automatically generate special methods like `__init__`, `__repr__`, etc., for classes.
- **`typing`**:
  - Various type hints to specify the expected types of variables, return types of functions, etc.
  - `Any`, `AsyncGenerator`, `Awaitable`, `Callable`, `List`, `Optional`, `TypedDict`, `cast`.
- **`urllib.parse.urljoin`**:
  - Joins a base URL and a relative URL to form an absolute URL.

Third-Party Libraries

- **`aiohttp`**:
  - Used for making asynchronous HTTP requests, which is crucial for non-blocking operations in an asynchronous application.
- **`azure.search.documents.aio.SearchClient`**:
  - An asynchronous client for interacting with Azure Cognitive Search.
- **`azure.search.documents.models`**:
  - **`QueryCaptionResult`**: Represents the result of a query caption.
  - **`QueryType`**: Enumerates the types of queries (e.g., semantic).

    - **`VectorizedQuery`, `VectorQuery`**: Represent vector-based queries for more advanced search operations.
- **`openai.AsyncOpenAI`**:
    - An asynchronous client for interacting with OpenAI's API.
- **`openai.types.chat.ChatCompletionMessageParam`**:
    - Represents the parameters for a chat completion message, useful for OpenAI's chat models.

Local Imports

- **`core.authentication.AuthenticationHelper`**:
    - Likely a helper class for managing authentication processes within the application.
- **`text.nonewlines`**:
    - A utility function to remove newline characters from a string, likely to clean up text before processing or displaying it.

Summary

This code sets up the necessary imports for a Python module that integrates Azure Cognitive Search with OpenAI's language models. It pulls in various utilities, clients, and type hints needed for asynchronous operations, search queries, and handling text. The use of abstract base classes suggests that this module will define base functionality that can be extended by subclasses to implement specific search and retrieval strategies. The presence of `AuthenticationHelper` implies that secure access to APIs will be managed within this module.

The `Document` class is a data structure used to represent a document retrieved from the Azure Cognitive Search. It's defined using the `@dataclass` decorator, which simplifies the class definition by automatically generating special methods like `__init__`, `__repr__`, and `__eq__`. Here's a detailed explanation of each attribute in the `Document` class:

Attributes:

1. **`id: Optional[str]`**
   - Represents the unique identifier of the document. It is optional, meaning it can be `None`.
2. **`content: Optional[str]`**
   - Contains the textual content of the document. This field is optional and can be `None`.
3. **`embedding: Optional[List[float]]`**
   - Holds a vector representation (embedding) of the document's content, typically used for similarity search. This list of floats is optional.
4. **`image_embedding: Optional[List[float]]`**
   - Similar to `embedding`, but specifically for images. It holds a vector representation of an image associated with the document. This is also optional.
5. **`category: Optional[str]`**
   - Specifies the category or classification of the document. This field is optional.
6. **`sourcepage: Optional[str]`**
   - Indicates the source page from where the document was retrieved. This is optional and can be `None`.
7. **`sourcefile: Optional[str]`**
   - Contains the file path or name of the source file. This field is optional.
8. **`oids: Optional[List[str]]`**

    - Represents a list of object identifiers (OIDs) associated with the document. This is optional and can be `None`.
9. **`groups: Optional[List[str]]`**
    - Specifies a list of groups or categories that the document belongs to. This field is optional.
10. **`captions: List[QueryCaptionResult]`**
    - Contains a list of captions generated from the document by the Azure Cognitive Search, which provides extractive summaries or highlights. This field is mandatory and not optional.
11. **`score: Optional[float] = None`**
    - Holds the search score assigned to the document by the Azure Cognitive Search. This field is optional and defaults to `None` if not provided.
12. **`reranker_score: Optional[float] = None`**
    - Contains the score assigned to the document by a reranker, if applicable. This field is optional and defaults to `None`.

Usage:

This `Document` class is designed to encapsulate all relevant information about a document retrieved from a search query. The optional nature of most fields provides flexibility, allowing documents to be represented even when some data is missing. The list of captions and scores helps in further processing and displaying search results with additional context and relevance ranking.

Here's a brief example of how you might use this `Document` class:

```
from azure.search.documents.models import
QueryCaptionResult
```

```
# Example usage

doc = Document(

    id="123",

    content="This is a sample document.",

    embedding=[0.1, 0.2, 0.3],

    category="Sample Category",

    sourcepage="page1",

    sourcefile="file1",

    oids=["oid1"],

    groups=["group1"],

    captions=[QueryCaptionResult(text="Sample caption",
highlights=[])],

    score=0.95,

    reranker_score=0.9

)


print(doc)
```

In this example, a `Document` object is created with some sample data, and its representation is printed out. The class structure makes it easy to handle and pass around document data within the application.

The `serialize_for_results` method is a member function of the `Document` class that converts the attributes of a `Document` instance into a dictionary format. This is useful for preparing the document data to be returned as a JSON response from an API or for other forms of data serialization.

Here's a detailed breakdown of what this method does:

Method Definition:

```
def serialize_for_results(self) -> dict[str, Any]:
```

- **Return Type**: The method returns a dictionary where the keys are strings and the values can be of any type (`dict[str, Any]`).

Method Body:

```
return {
    "id": self.id,
    "content": self.content,
    "embedding":
Document.trim_embedding(self.embedding),
    "imageEmbedding":
Document.trim_embedding(self.image_embedding),
    "category": self.category,
    "sourcepage": self.sourcepage,
    "sourcefile": self.sourcefile,
    "oids": self.oids,
```

```
    "groups": self.groups,
    "captions": (
        [
            {
                "additional_properties":
caption.additional_properties,
                "text": caption.text,
                "highlights": caption.highlights,
            }
            for caption in self.captions
        ]
        if self.captions
        else []
    ),
    "score": self.score,
    "reranker_score": self.reranker_score,
}
```

- **`"id": self.id`**: Directly takes the `id` attribute of the `Document` instance.
- **`"content": self.content`**: Directly takes the `content` attribute.

- `"embedding": Document.trim_embedding(self.embedding)`: Uses the `trim_embedding` class method to process the `embedding` attribute before including it in the dictionary.
- `"imageEmbedding": Document.trim_embedding(self.image_embedding)`: Similarly processes the `image_embedding` attribute using `trim_embedding`.
- `"category": self.category`: Directly takes the `category` attribute.
- `"sourcepage": self.sourcepage`: Directly takes the `sourcepage` attribute.
- `"sourcefile": self.sourcefile`: Directly takes the `sourcefile` attribute.
- `"oids": self.oids`: Directly takes the `oids` attribute.
- `"groups": self.groups`: Directly takes the `groups` attribute.
- `"captions"`: Processes the `captions` attribute, which is a list of `QueryCaptionResult` objects. It converts each caption into a dictionary with `additional_properties`, `text`, and `highlights`.
    - `for caption in self.captions`: Iterates over each caption in the `captions` list.
    - `if self.captions else []`: If `captions` is not empty, it constructs the list of caption dictionaries; otherwise, it returns an empty list.
- `"score": self.score`: Directly takes the `score` attribute.
- `"reranker_score": self.reranker_score`: Directly takes the `reranker_score` attribute.

`trim_embedding` Method:

The `trim_embedding` method is used to format the `embedding` and `image_embedding` lists. If these lists are longer than two elements, it truncates them and provides a summary. This helps in reducing the amount of data sent in the response, making it more readable and manageable.

```
@classmethod

def trim_embedding(cls, embedding:
Optional[List[float]]) -> Optional[str]:

    """Returns a trimmed list of floats from the vector
embedding."""

    if embedding:

        if len(embedding) > 2:

            # Format the embedding list to show the
first 2 items followed by the count of the remaining
items.

            return f"[{embedding[0]}, {embedding[1]}
...+{len(embedding) - 2} more]"

        else:

            return str(embedding)

    return None
```

- **`if embedding`**: Checks if the embedding list is not `None`.

- `if len(embedding) > 2`: If the list has more than two elements, it formats the first two elements and indicates the count of the remaining elements.
- `return str(embedding)`: If the list has two or fewer elements, it converts the list to a string.
- `return None`: If the embedding is `None`, it returns `None`.

Summary

The `serialize_for_results` method converts a `Document` instance into a dictionary, suitable for JSON serialization. It processes each attribute of the document, directly including some and formatting others (like embeddings and captions) to make the resulting dictionary concise and readable. This method is essential for preparing document data to be returned from APIs or stored in a structured format.

The `trim_embedding` method is a class method within the `Document` class. Its purpose is to format the `embedding` or `image_embedding` attributes into a more concise string representation, particularly useful when these lists can be very large. Here's a detailed explanation:

Method Definition:

```
@classmethod

def trim_embedding(cls, embedding:
Optional[List[float]]) -> Optional[str]:
```

- **`@classmethod`**: This decorator indicates that `trim_embedding` is a class method, meaning it receives the class (`cls`) as the first argument rather than an instance of the class. This allows it to be called on the class itself, not just on instances.
- **`embedding: Optional[List[float]]`**: The method takes a single argument, `embedding`, which is an optional list of floats.
- **`-> Optional[str]`**: The method returns an optional string. It can return either a formatted string or `None`.

Method Body:

```
"""Returns a trimmed list of floats from the vector
embedding."""
```

- The docstring briefly describes the purpose of the method.

```
if embedding:
```

- Checks if `embedding` is not `None`. If `embedding` is `None`, the method will skip to the final return statement and return `None`.

```
if len(embedding) > 2:

    # Format the embedding list to show the first 2
items followed by the count of the remaining items.

    return f"[{embedding[0]}, {embedding[1]}
...+{len(embedding) - 2} more]"
```

- **`if len(embedding) > 2`**: Checks if the length of the `embedding` list is greater than 2.
  - **`return f"[{embedding[0]}, {embedding[1]} ...+{len(embedding) - 2} more]"`**: If the length is greater than 2, it formats the first two elements of the list and indicates how many additional elements there are. This helps in providing a summary view of potentially large embeddings.

```
else:

    return str(embedding)
```

- **`else`**: If the length of the `embedding` list is 2 or fewer elements:
  - **`return str(embedding)`**: Converts the list to a string and returns it. This way, the entire embedding is shown if it is short enough to be easily readable.

```
return None
```

- If `embedding` is `None`, the method returns `None`.

Summary:

The `trim_embedding` method provides a way to convert potentially long lists of float values (used as embeddings) into a concise string representation. This is useful for logging, displaying, or returning the embeddings in an API response without overwhelming the consumer with large amounts of data.

Example Usage:

Here's an example of how the `trim_embedding` method works with different lengths of embedding lists:

```
# Example embeddings

short_embedding = [0.1, 0.2]

long_embedding = [0.1, 0.2, 0.3, 0.4, 0.5]


# Using the trim_embedding method

print(Document.trim_embedding(short_embedding))  #
Output: "[0.1, 0.2]"

print(Document.trim_embedding(long_embedding))   #
Output: "[0.1, 0.2 ...+3 more]"

print(Document.trim_embedding(None))             #
Output: None
```

In this example:

- For a short embedding of two elements, it returns the string representation of the entire list.
- For a longer embedding, it returns a summary showing the first two elements followed by the count of the remaining elements.
- For a `None` embedding, it returns `None`.

The provided code includes two parts: the `ThoughtStep` dataclass and the `Approach` abstract base class (ABC). Let's break down each part in detail.

ThoughtStep Dataclass

The `ThoughtStep` class is a simple data structure defined using the `@dataclass` decorator. This decorator automatically generates special methods like `__init__`, `__repr__`, and `__eq__` for the class based on its fields.

```
@dataclass

class ThoughtStep:

    title: str

    description: Optional[Any]

    props: Optional[dict[str, Any]] = None
```

Attributes:

- **`title: str`**
  - A string representing the title of the thought step. This is a mandatory field.
- **`description: Optional[Any]`**
  - An optional field that can hold a description of any type. If no description is provided, it defaults to `None`.
- **`props: Optional[dict[str, Any]] = None`**
  - An optional dictionary to store additional properties related to the thought step. It defaults to `None` if not provided.

Approach Abstract Base Class

The `Approach` class is an abstract base class (ABC) that provides a template for different approaches that can be implemented in the application. It uses the `ABC` module to define abstract methods that must be implemented by subclasses.

`__init__` Method

The `__init__` method initializes an instance of the `Approach` class with several attributes.

```
class Approach(ABC):

    def __init__(

        self,

        search_client: SearchClient,

        openai_client: AsyncOpenAI,

        auth_helper: AuthenticationHelper,

        query_language: Optional[str],

        query_speller: Optional[str],

        embedding_deployment: Optional[str],  # Not
needed for non-Azure OpenAI or for
retrieval_mode="text"

        embedding_model: str,

        embedding_dimensions: int,

        openai_host: str,
```

```
        vision_endpoint: str,
        vision_token_provider: Callable[[], Awaitable[str]],
    ):
        self.search_client = search_client
        self.openai_client = openai_client
        self.auth_helper = auth_helper
        self.query_language = query_language
        self.query_speller = query_speller
        self.embedding_deployment = embedding_deployment
        self.embedding_model = embedding_model
        self.embedding_dimensions = embedding_dimensions
        self.openai_host = openai_host
        self.vision_endpoint = vision_endpoint
        self.vision_token_provider = vision_token_provider
```

Parameters:

- **`search_client: SearchClient`**
  - An instance of `SearchClient` used to interact with the Azure Cognitive Search service.
- **`openai_client: AsyncOpenAI`**
  - An instance of `AsyncOpenAI` used to interact with OpenAI's API.
- **`auth_helper: AuthenticationHelper`**
  - An instance of `AuthenticationHelper` used for handling authentication and authorization.
- **`query_language: Optional[str]`**
  - An optional string specifying the language of the query.
- **`query_speller: Optional[str]`**
  - An optional string specifying the speller configuration for the query.
- **`embedding_deployment: Optional[str]`**
  - An optional string specifying the deployment of the embedding model. This is not needed for non-Azure OpenAI or for `retrieval_mode="text"`.
- **`embedding_model: str`**
  - A string specifying the embedding model to use.
- **`embedding_dimensions: int`**
  - An integer specifying the dimensions of the embedding vector.
- **`openai_host: str`**
  - A string specifying the OpenAI host.
- **`vision_endpoint: str`**
  - A string specifying the endpoint for the vision service.
- **`vision_token_provider: Callable[[], Awaitable[str]]`**
  - A callable that returns an awaitable, which provides a token for accessing the vision service.

`build_filter` Method

The `build_filter` method constructs a filter string based on the provided overrides and authentication claims.

```
def build_filter(self, overrides: dict[str, Any], auth_claims: dict[str, Any]) -> Optional[str]:

    exclude_category = overrides.get("exclude_category")

    security_filter = self.auth_helper.build_security_filters(overrides, auth_claims)

    filters = []

    if exclude_category:

        filters.append("category ne '{}'".format(exclude_category.replace("'", "''")))

    if security_filter:

        filters.append(security_filter)

    return None if len(filters) == 0 else " and ".join(filters)
```

Parameters:

- **`overrides: dict[str, Any]`**
  - A dictionary of override parameters that can modify the behavior of the filter.
- **`auth_claims: dict[str, Any]`**

    - A dictionary of authentication claims that may be used to build security filters.

Functionality:

- **`exclude_category = overrides.get("exclude_category")`**
    - Retrieves the `exclude_category` from the overrides dictionary, if it exists.
- **`security_filter = self.auth_helper.build_security_filters(overrides, auth_claims)`**
    - Calls `build_security_filters` on the `auth_helper` instance to get the security filter string.
- **`filters = []`**
    - Initializes an empty list to store filter conditions.
- **`if exclude_category`**
    - If `exclude_category` is provided, it adds a filter to exclude documents from the specified category.
- **`if security_filter`**
    - If a security filter is provided, it adds it to the filters list.
- **`return None if len(filters) == 0 else " and ".join(filters)`**
    - If there are no filters, returns `None`. Otherwise, joins the filters with `and` and returns the resulting string.

Summary

- **`ThoughtStep` Dataclass**: A simple structure to hold a thought step's title, description, and optional properties.
- **`Approach` Abstract Base Class**: Provides a template for implementing different approaches, including initialization with several services and clients, and a method for building filters based on overrides and authentication claims.

The provided code includes several methods and functionalities in an abstract class `Approach`, designed to work with Azure Cognitive Search and OpenAI's services. Here's an explanation of each method:

`search` Method

This asynchronous method performs a search using the Azure Cognitive Search service.

python

Copy code

```
async def search(
    self,
    top: int,
    query_text: Optional[str],
    filter: Optional[str],
    vectors: List[VectorQuery],
    use_semantic_ranker: bool,
    use_semantic_captions: bool,
    minimum_search_score: Optional[float],
    minimum_reranker_score: Optional[float],
) -> List[Document]:
```

Parameters:

- `top: int`: The maximum number of search results to return.
- `query_text: Optional[str]`: The search query text.
- `filter: Optional[str]`: The filter string to apply to the search.
- `vectors: List[VectorQuery]`: A list of vector queries for vector-based search.
- `use_semantic_ranker: bool`: Whether to use semantic ranking.
- `use_semantic_captions: bool`: Whether to use semantic captions.
- `minimum_search_score: Optional[float]`: Minimum search score for a document to be considered.
- `minimum_reranker_score: Optional[float]`: Minimum reranker score for a document to be considered.

Functionality:

1. **Perform Search**:
    - If `use_semantic_ranker` is `True` and `query_text` is provided, it performs a semantic search using the `SearchClient`.
    - Otherwise, it performs a standard search.
2. **Collect Results**:
    - Iterates through the search results pages and collects documents into a list.
3. **Filter Results**:
    - Filters documents based on `minimum_search_score` and `minimum_reranker_score`.
4. **Return Results**:
    - Returns the filtered list of documents.

`get_sources_content` Method

This method extracts and formats the content of the sources from the search results.

```
def get_sources_content(

    self, results: List[Document],
use_semantic_captions: bool, use_image_citation: bool

) -> list[str]:
```

Parameters:

- **`results: List[Document]`**: The list of documents from search results.
- **`use_semantic_captions: bool`**: Whether to use semantic captions.
- **`use_image_citation: bool`**: Whether to use image citation format.

Functionality:

1. **With Semantic Captions**:
    - If `use_semantic_captions` is `True`, it formats the source content using the captions from the documents.
2. **Without Semantic Captions**:
    - If `use_semantic_captions` is `False`, it formats the source content using the document content.
3. **Return Formatted Content**:
    - Returns the formatted source content as a list of strings.

`get_citation` Method

This method generates a citation for a given source page.

```
def get_citation(self, sourcepage: str,
use_image_citation: bool) -> str:
```

Parameters:

- **`sourcepage: str`**: The source page string.
- **`use_image_citation: bool`**: Whether to use image citation format.

Functionality:

1. **Image Citation**:
    - If `use_image_citation` is `True`, it returns the `sourcepage` as-is.
2. **Non-Image Citation**:
    - If `use_image_citation` is `False`, it checks if the `sourcepage` is a PNG file.
    - If it is, it converts the PNG file reference to a corresponding PDF page reference.
    - Otherwise, it returns the `sourcepage`.

`compute_text_embedding` Method

This asynchronous method computes a text embedding using the OpenAI API.

```
async def compute_text_embedding(self, q: str):
```

Parameters:

- `q: str`: The text query for which to compute the embedding.

Functionality:

1. **Supported Dimensions**:
    - A dictionary `SUPPORTED_DIMENSIONS_MODEL` specifies whether a model supports custom dimensions.
2. **Dimensions Arguments**:
    - Constructs arguments for dimensions based on the model.
3. **Request Embedding**:
    - Requests the text embedding from the OpenAI API using the specified model.
4. **Return Vectorized Query**:
    - Returns a `VectorizedQuery` object with the computed vector.

`compute_image_embedding` Method

This asynchronous method computes an image embedding using the Azure Vision API.

```
async def compute_image_embedding(self, q: str):
```

Parameters:

- `q: str`: The text query for which to compute the image embedding.

Functionality:

1. **Endpoint and Headers**:
    - Constructs the endpoint URL and headers for the API request.
2. **Authorization**:
    - Adds the authorization token to the headers.
3. **Request Embedding**:

    - Sends a request to the Vision API to compute the image embedding.
4. **Return Vectorized Query**:
    - Returns a `VectorizedQuery` object with the computed vector.

`run` Method

This abstract method must be implemented by subclasses. It defines how to run the approach with a list of messages and optional state/context.

```
async def run(
    self,
    messages: list[ChatCompletionMessageParam],
    session_state: Any = None,
    context: dict[str, Any] = {},
) -> dict[str, Any]:
    raise NotImplementedError
```

`run_stream` Method

This abstract method must be implemented by subclasses. It defines how to run the approach in a streaming fashion with a list of messages and optional state/context.

python

Copy code

```
async def run_stream(
```

```
        self,
        messages: list[ChatCompletionMessageParam],
        session_state: Any = None,
        context: dict[str, Any] = {},
    ) -> AsyncGenerator[dict[str, Any], None]:
        raise NotImplementedError
```

Summary

The `Approach` abstract class provides a framework for building search and retrieval solutions using Azure Cognitive Search and OpenAI's services. It includes methods for performing searches, computing embeddings, extracting and formatting source content, and generating citations. Subclasses of `Approach` must implement the `run` and `run_stream` methods to define specific behaviors for different approaches.