Vectors and Vector Search

You're absolutely right. I missed including the critical vector search and embedding model configuration, which is essential for both semantic and hybrid search in Azure AI Search. Let me add the necessary components to properly define and use embedding models for both indexing and query vectorization.

Vector Search and Embedding Model Configuration in Azure AI Search

1. Enhanced Index Schema with Vector Search Configuration

First, let's update our index creation to include vector search capabilities:

```
from azure.search.documents.indexes.models import (
    SearchIndex,
    SearchField,
    SearchFieldDataType,
    SimpleField,
    SearchableField,
    VectorSearch,
    VectorSearchProfile,
    VectorSearchAlgorithmConfiguration,
    HnswAlgorithmConfiguration,
    VectorSearchDimensions
)

def create_vector_enabled_index(index_client, index_name):
    """Create an index with vector search capabilities"""

    # Define the fields, including vector fields for embeddings
    fields = [
        # Key field (required)
        SimpleField(name="id", type=SearchFieldDataType.String, key=True),

        # Core searchable content
        SearchableField(name="title", type=SearchFieldDataType.String,
                       analyzer_name="en.microsoft"),
        SearchableField(name="content", type=SearchFieldDataType.String,
                       analyzer_name="en.microsoft"),

        # Vector embeddings field for content
        SimpleField(
            name="content_vector",

type=SearchFieldDataType.Collection(SearchFieldDataType.Single),
            vector_search_dimensions=1536,  # OpenAI ada-002 embedding
size
            vector_search_profile_name="default-profile"
        ),

        # Vector embeddings field for title
```

```
        SimpleField(
            name="title_vector",

type=SearchFieldDataType.Collection(SearchFieldDataType.Single),
            vector_search_dimensions=1536,  # OpenAI ada-002 embedding
size
            vector_search_profile_name="default-profile"
        ),

        # Metadata fields (as in the previous example)
        SimpleField(name="created_date",
type=SearchFieldDataType.DateTimeOffset,
                   filterable=True, sortable=True),
        SimpleField(name="last_updated",
type=SearchFieldDataType.DateTimeOffset,
                   filterable=True, sortable=True),
        SimpleField(name="author", type=SearchFieldDataType.String,
                   filterable=True, facetable=True),
        SimpleField(name="document_type", type=SearchFieldDataType.String,
                   filterable=True, facetable=True),
        SimpleField(name="department", type=SearchFieldDataType.String,
                   filterable=True, facetable=True),
        SimpleField(name="priority", type=SearchFieldDataType.Int32,
                   filterable=True, sortable=True),
        SimpleField(name="status", type=SearchFieldDataType.String,
                   filterable=True, facetable=True),
        SimpleField(name="tags",
type=SearchFieldDataType.Collection(SearchFieldDataType.String),
                   filterable=True, facetable=True),
        SimpleField(name="geo_location",
type=SearchFieldDataType.GeographyPoint,
                   filterable=True),
        SimpleField(name="view_count", type=SearchFieldDataType.Int32,
                   filterable=True, sortable=True),
        SimpleField(name="relevance_score",
type=SearchFieldDataType.Double,
                   filterable=True, sortable=True)
    ]

    # Configure vector search
    vector_search = VectorSearch(
        algorithms=[
            VectorSearchAlgorithmConfiguration(
                name="default-algorithm",
                kind="hnsw",
                hnsw_parameters=HnswAlgorithmConfiguration(
                    m=4,  # Number of connections per node
                    ef_construction=400,  # Size of the dynamic list for
nearest neighbors
                    ef_search=500,  # Size of the dynamic list for
searching
                    metric="cosine"  # Distance metric (cosine, euclidean,
dotProduct)
                )
```

```
            )
        ],
        profiles=[
            VectorSearchProfile(
                name="default-profile",
                algorithm_configuration_name="default-algorithm"
            )
        ]
    )

    # Create the index with vector search configuration
    index = SearchIndex(
        name=index_name,
        fields=fields,
        vector_search=vector_search
    )

    try:
        result = index_client.create_or_update_index(index)
        print(f"Vector-enabled index '{index_name}' created
successfully.")
        return result
    except Exception as e:
        print(f"Error creating vector-enabled index: {str(e)}")
        raise
```

2. Generating Embeddings with OpenAI for Documents

Now, let's add functionality to generate embeddings for our documents using OpenAI:

```
import openai
import numpy as np
from tenacity import retry, stop_after_attempt, wait_random_exponential

def setup_openai_client(api_key):
    """Setup the OpenAI client with API key"""
    openai.api_key = api_key
    print("OpenAI client configured successfully")

@retry(wait=wait_random_exponential(min=1, max=20),
stop=stop_after_attempt(6))
def generate_embeddings(text, model="text-embedding-ada-002"):
    """
    Generate embeddings for the given text using OpenAI's embedding model
    with retry logic for robustness
    """
    try:
        response = openai.embeddings.create(
            model=model,
            input=text
        )
```

```
        return response.data[0].embedding
    except Exception as e:
        print(f"Error generating embeddings: {str(e)}")
        raise
```

3. Uploading Documents with Embeddings

Now, let's modify our document upload function to include embeddings:

```
def upload_documents_with_embeddings(search_client, openai_api_key):
    """Upload sample documents with embeddings and metadata"""

    # Configure OpenAI client
    setup_openai_client(openai_api_key)

    # Current time for timestamps
    from datetime import datetime, timedelta
    current_time = datetime.utcnow().isoformat()
    yesterday = (datetime.utcnow() - timedelta(days=1)).isoformat()

    # Sample documents
    documents_base = [
        {
            "id": "doc-001",
            "title": "Azure AI Search Implementation Guide",
            "content": "This comprehensive guide covers advanced
implementation techniques for Azure AI Search, including hybrid search
models, vector search, and semantic ranking.",
            "created_date": yesterday,
            "last_updated": current_time,
            "author": "Alice Johnson",
            "document_type": "technical_guide",
            "department": "Cloud Services",
            "priority": 1,
            "status": "published",
            "tags": ["azure", "ai", "search", "hybrid", "implementation"],
            "geo_location": {
                "type": "Point",
                "coordinates": [-122.12, 47.67]  # Seattle coordinates
            },
            "view_count": 1200,
            "relevance_score": 0.95
        },
        {
            "id": "doc-002",
            "title": "Python Tools for Azure Integration",
            "content": "Learn how to use Python SDKs and libraries to
integrate with Azure services, including Azure AI Search, Cognitive
Services, and Azure Machine Learning.",
            "created_date": yesterday,
            "last_updated": current_time,
```

```
            "author": "Bob Smith",
            "document_type": "tutorial",
            "department": "Development",
            "priority": 2,
            "status": "published",
            "tags": ["python", "azure", "integration", "sdk"],
            "geo_location": {
                "type": "Point",
                "coordinates": [-74.01, 40.71]  # New York coordinates
            },
            "view_count": 850,
            "relevance_score": 0.88
        },
        {
            "id": "doc-003",
            "title": "Natural Language Processing with Azure",
            "content": "This document explores how to implement NLP
solutions using Azure's AI services, focusing on text analysis, sentiment
detection, and knowledge mining capabilities.",
            "created_date": current_time,
            "last_updated": current_time,
            "author": "Carol Davis",
            "document_type": "whitepaper",
            "department": "AI Research",
            "priority": 1,
            "status": "draft",
            "tags": ["nlp", "azure", "ai", "text-analysis", "sentiment"],
            "geo_location": {
                "type": "Point",
                "coordinates": [-118.24, 34.05]  # Los Angeles coordinates
            },
            "view_count": 320,
            "relevance_score": 0.92
        }
    ]

    # Add embeddings to each document
    documents = []
    for doc in documents_base:
        # Generate embeddings for title and content
        title_embedding = generate_embeddings(doc["title"])
        content_embedding = generate_embeddings(doc["content"])

        # Add embeddings to the document
        doc["title_vector"] = title_embedding
        doc["content_vector"] = content_embedding

        documents.append(doc)

    try:
        result = search_client.upload_documents(documents=documents)
        print(f"Uploaded {len(result)} documents with embeddings and
metadata")
        return True
```

```
    except Exception as e:
        print(f"Error uploading documents with embeddings: {str(e)}")
        return False
```

4. Performing Vector Search and Hybrid Search

Now, let's implement functions for pure vector search, hybrid search, and semantic hybrid search:

```
def perform_vector_search(search_client, query_text, openai_api_key,
vector_field="content_vector"):
    """
    Perform a pure vector search using embeddings
    """
    # Generate embeddings for the query
    setup_openai_client(openai_api_key)
    query_vector = generate_embeddings(query_text)

    # Define search options for vector search
    vector_search_options = {
        "vector": query_vector,
        "vector_fields": [vector_field],
        "top": 10,
        "select": "id,title,content,author,document_type,department,tags"
    }

    try:
        # Execute the vector search
        results = search_client.search(search_text=None,
**vector_search_options)

        # Process the results
        print(f"\nVector search results for query: '{query_text}'")

        count = 0
        for result in results:
            count += 1
            print(f"\nDocument ID: {result['id']}")
            print(f"Title: {result['title']}")
            print(f"Score: {result['@search.score']}")
            print(f"Author: {result['author']}")
            print(f"Document Type: {result['document_type']}")
            print(f"Department: {result['department']}")
            print(f"Tags: {', '.join(result['tags'])}")
            print(f"Content (snippet): {result['content'][:150]}...")

        print(f"\nFound {count} documents")

    except Exception as e:
        print(f"Vector search error: {str(e)}")

def perform_hybrid_vector_text_search(search_client, query_text,
```

```
openai_api_key, options=None):
    """
    Perform a hybrid search combining traditional keyword search with
vector search
    """
    if options is None:
        options = {}

    # Generate embeddings for the query
    setup_openai_client(openai_api_key)
    query_vector = generate_embeddings(query_text)

    # Default search options for hybrid search
    search_options = {
        "search_text": query_text,  # Traditional keyword search
        "vector": query_vector,     # Vector search component
        "vector_fields": ["content_vector"],  # Field to perform vector
search on
        "top": 10,
        "select": "*",
        "include_total_count": True,
        "highlight_fields": "content",
        "highlight_pre_tag": "<b>",
        "highlight_post_tag": "</b>"
    }

    # Update with any user-provided options
    search_options.update(options)

    try:
        # Execute the hybrid search
        results = search_client.search(**search_options)

        # Process the results
        print(f"\nHybrid vector + text search results for:
'{query_text}'")

        count = 0
        for result in results:
            count += 1
            print(f"\nDocument ID: {result['id']}")
            print(f"Title: {result['title']}")
            print(f"Score: {result['@search.score']}")
            print(f"Author: {result['author']}")
            print(f"Document Type: {result['document_type']}")

            # Display highlighted content if available
            if '@search.highlights' in result and 'content' in
result['@search.highlights']:
                print("Content highlights:")
                for highlight in result['@search.highlights']['content']:
                    print(f"  ...{highlight}...")

        print(f"\nFound {count} documents")
```

```
    except Exception as e:
        print(f"Hybrid search error: {str(e)}")

def perform_semantic_hybrid_search(search_client, query_text,
openai_api_key, options=None):
    """
    Perform a semantic hybrid search that combines vector similarity with
    traditional keyword search and semantic reranking
    """
    if options is None:
        options = {}

    # Generate embeddings for the query
    setup_openai_client(openai_api_key)
    query_vector = generate_embeddings(query_text)

    # Default search options for semantic hybrid search
    search_options = {
        "search_text": query_text,  # Traditional keyword search
        "vector": query_vector,     # Vector search component
        "vector_fields": ["content_vector"],  # Field to perform vector
search on
        "query_type": "semantic",   # Enable semantic search
        "semantic_configuration_name": "default-semantic-config",  # Must
be configured on your search service
        "query_caption": "extractive|highlight-false",  # Extract captions
from the content
        "query_answer": "extractive|count-1",  # Extract answers
        "top": 10,
        "select": "*",
        "include_total_count": True
    }

    # Update with any user-provided options
    search_options.update(options)

    try:
        # Execute the semantic hybrid search
        results = search_client.search(**search_options)

        # Process the results
        print(f"\nSemantic hybrid search results for: '{query_text}'")

        count = 0
        for result in results:
            count += 1
            print(f"\nDocument ID: {result['id']}")
            print(f"Title: {result['title']}")
            print(f"Score: {result['@search.score']}")

            # Display semantic information if available
            if '@search.rerankerScore' in result:
                print(f"Reranker Score:
```

```
{result['@search.rerankerScore']}")

            if '@search.captions' in result:
                print("Captions:")
                for caption in result['@search.captions']:
                    print(f"  {caption['text']}")

            if '@search.answers' in result:
                print("Answers:")
                for answer in result['@search.answers']:
                    print(f"  {answer['text']}")

        print(f"\nFound {count} documents")

    except Exception as e:
        print(f"Semantic hybrid search error: {str(e)}")
```

5. Creating a Semantic Configuration

To fully utilize semantic search capabilities, we need to create a semantic configuration:

```
from azure.search.documents.indexes.models import (
    SemanticConfiguration,
    SemanticField,
    SemanticSettings,
    PrioritizedFields
)

def add_semantic_configuration(index_client, index_name):
    """Add semantic configuration to an existing index"""

    try:
        # Get the existing index
        index = index_client.get_index(index_name)

        # Define semantic configuration
        semantic_config = SemanticConfiguration(
            name="default-semantic-config",
            prioritized_fields=PrioritizedFields(
                title_field=SemanticField(field_name="title"),
                prioritized_content_fields=[
                    SemanticField(field_name="content")
                ],
                prioritized_keywords_fields=[
                    SemanticField(field_name="tags")
                ]
            )
        )

        # Create semantic settings with the configuration
        semantic_settings = SemanticSettings(
```

```
            configurations=[semantic_config]
        )

        # Add semantic settings to the index
        index.semantic_settings = semantic_settings

        # Update the index
        result = index_client.create_or_update_index(index)
        print(f"Added semantic configuration to index '{index_name}'")
        return result

    except Exception as e:
        print(f"Error adding semantic configuration: {str(e)}")
        raise
```

6. Using Azure OpenAI for Embedding Generation

Alternatively, if you're using Azure OpenAI rather than the public OpenAI API, you can use this function:

```
import openai
from tenacity import retry, stop_after_attempt, wait_random_exponential

def setup_azure_openai_client(api_key, api_base, api_version="2023-05-15"):
    """Setup the Azure OpenAI client"""
    openai.api_type = "azure"
    openai.api_key = api_key
    openai.api_base = api_base
    openai.api_version = api_version
    print("Azure OpenAI client configured successfully")

@retry(wait=wait_random_exponential(min=1, max=20), stop=stop_after_attempt(6))
def generate_azure_openai_embeddings(text, deployment_name):
    """
    Generate embeddings using Azure OpenAI
    """
    try:
        response = openai.embeddings.create(
            input=text,
            deployment_id=deployment_name
        )
        return response.data[0].embedding
    except Exception as e:
        print(f"Error generating Azure OpenAI embeddings: {str(e)}")
        raise
```

7. Complete Implementation with All Search Types

Here's a complete example bringing together all the search types:

```
def main():
    """Main function to demonstrate Azure AI Search with embeddings for 
vector and hybrid search"""
    try:
        # Authenticate
        credential = AzureCliCredential()

        # Get subscription ID
        from azure.mgmt.resource import SubscriptionClient
        subscription_client = SubscriptionClient(credential)
        subscription = next(subscription_client.subscriptions.list())
        subscription_id = subscription.subscription_id

        print(f"Using subscription: {subscription.display_name} 
({subscription_id})")

        # Get search service details
        resource_group_name = input("Enter your resource group name: ")
        service_name = input("Enter your search service name: ")

        # Get OpenAI API key for embeddings
        openai_api_key = input("Enter your OpenAI API key for generating 
embeddings: ")

        # Create Search Management client
        search_mgmt_client = SearchManagementClient(credential, 
subscription_id)

        # Get the admin key
        try:
            admin_key = search_mgmt_client.admin_keys.get(
                resource_group_name=resource_group_name,
                search_service_name=service_name
            ).primary_key

            print(f"Successfully connected to search service: 
{service_name}")

            # Set up search clients
            service_endpoint = 
f"https://{service_name}.search.windows.net"
            credential = AzureKeyCredential(admin_key)

            # Create index client
            index_client = SearchIndexClient(endpoint=service_endpoint, 
credential=credential)

            # Create a new vector-enabled index
            index_name = "vector-hybrid-index"
            create_vector_enabled_index(index_client, index_name)

            # Add semantic configuration
```

```
            add_semantic_configuration(index_client, index_name)

            # Create a search client
            search_client = SearchClient(endpoint=service_endpoint,
index_name=index_name, credential=credential)

            # Upload documents with embeddings
            upload_documents_with_embeddings(search_client,
openai_api_key)

            # Demonstrate different search types
            demonstrate_search_types(search_client, openai_api_key)

            # Interactive search mode
            interactive_search_demo(search_client, openai_api_key)

        except ResourceNotFoundError:
            print(f"Error: Search service '{service_name}' not found in
resource group '{resource_group_name}'")

    except Exception as e:
        print(f"An error occurred: {str(e)}")

def demonstrate_search_types(search_client, openai_api_key):
    """Demonstrate different search types with the same query"""

    # Sample query
    query = "azure machine learning implementation"

    print("\n=== COMPARING DIFFERENT SEARCH TYPES ===")
    print(f"Query: '{query}'")

    # 1. Pure text search (keyword-based)
    print("\n--- TRADITIONAL KEYWORD SEARCH ---")
    search_client.search(query, select="id,title,content,author",
include_total_count=True)

    # 2. Pure vector search
    print("\n--- PURE VECTOR SEARCH ---")
    perform_vector_search(search_client, query, openai_api_key)

    # 3. Hybrid search (text + vector)
    print("\n--- HYBRID SEARCH (TEXT + VECTOR) ---")
    perform_hybrid_vector_text_search(search_client, query,
openai_api_key)

    # 4. Semantic hybrid search
    print("\n--- SEMANTIC HYBRID SEARCH ---")
    perform_semantic_hybrid_search(search_client, query, openai_api_key)

    # 5. Filtered hybrid search
    print("\n--- FILTERED HYBRID SEARCH ---")
    perform_hybrid_vector_text_search(
        search_client,
```

```
        query,
        openai_api_key,
        {
            "filter": "document_type eq 'technical_guide' or tags/any(t: t
eq 'implementation')"
        }
    )

def interactive_search_demo(search_client, openai_api_key):
    """Interactive demo to try different search types"""

    print("\n=== INTERACTIVE SEARCH DEMO ===")
    print("Try different search types with your own queries. Enter 'exit'
to quit.")

    while True:
        query = input("\nEnter your search query: ")
        if query.lower() == 'exit':
            break

        print("\nSelect search type:")
        print("1. Traditional keyword search")
        print("2. Pure vector search")
        print("3. Hybrid search (text + vector)")
        print("4. Semantic hybrid search")
        print("5. Filtered hybrid search")

        choice = input("Enter your choice (1-5): ")

        if choice == '1':
            # Traditional keyword search
            results = search_client.search(
                query,
                select="id,title,content,author",
                include_total_count=True,
                highlight_fields="content"
            )
            display_results(results, "Traditional Keyword Search")

        elif choice == '2':
            # Pure vector search
            perform_vector_search(search_client, query, openai_api_key)

        elif choice == '3':
            # Hybrid search
            perform_hybrid_vector_text_search(search_client, query,
openai_api_key)

        elif choice == '4':
            # Semantic hybrid search
            perform_semantic_hybrid_search(search_client, query,
openai_api_key)

        elif choice == '5':
```

```
            # Filtered hybrid search
            filter_type = input("\nSelect filter type:\n1. Document type\n2. Department\n3. Tags\nEnter choice: ")

            filter_expr = None
            if filter_type == '1':
                doc_type = input("Enter document type to filter by: ")
                filter_expr = f"document_type eq '{doc_type}'"
            elif filter_type == '2':
                dept = input("Enter department to filter by: ")
                filter_expr = f"department eq '{dept}'"
            elif filter_type == '3':
                tag = input("Enter tag to filter by: ")
                filter_expr = f"tags/any(t: t eq '{tag}')"

            if filter_expr:
                perform_hybrid_vector_text_search(
                    search_client,
                    query,
                    openai_api_key,
                    {"filter": filter_expr}
                )
            else:
                print("No valid filter selected, performing standard hybrid search.")
                perform_hybrid_vector_text_search(search_client, query, openai_api_key)

        else:
            print("Invalid choice. Please try again.")

def display_results(results, search_type):
    """Display search results in a consistent format"""

    count = 0
    print(f"\n{search_type} results:")

    for result in results:
        count += 1
        print(f"\n{count}. {result['title']}")
        print(f"   ID: {result['id']}")
        print(f"   Score: {result['@search.score']}")

        # Display highlights if available
        if '@search.highlights' in result and 'content' in result['@search.highlights']:
            print("   Highlights:")
            for highlight in result['@search.highlights']['content']:
                print(f"   ...{highlight}...")

    print(f"\nFound {count} documents")

if __name__ == "__main__":
    main()
```

8. Using Azure Cognitive Search's Built-in Vectorization API

As an alternative to OpenAI for generating embeddings, you can use Azure Cognitive Search's built-in vectorization capabilities if you're using a version that supports it:

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

def perform_search_with_built_in_vectorization(search_client, query_text):
    """
    Perform search using Azure Cognitive Search's built-in vectorization API
    (requires a supported version of the service)
    """

    # Create a vectorizable text query
    vectorizable_query = VectorizableTextQuery(text=query_text, k=10, fields=["content"])

    # Define search options
    search_options = {
        "search_text": query_text,  # For keyword search component
        "vectorizable_text": vectorizable_query,  # For vector search with built-in vectorization
        "select": "*",
        "include_total_count": True
    }

    try:
        # Execute the search
        results = search_client.search(**search_options)

        # Process the results
        print(f"\nSearch with built-in vectorization for: '{query_text}'")

        count = 0
        for result in results:
            count += 1
            print(f"\nDocument ID: {result['id']}")
            print(f"Title: {result['title']}")
            print(f"Score: {result['@search.score']}")

        print(f"\nFound {count} documents")

    except Exception as e:
        print(f"Built-in vectorization search error: {str(e)}")
        print("Note: This feature requires Azure Cognitive Search service that supports built-in vectorization.")
```

9. Best Practices for Embedding Models in Hybrid Search

Here are best practices specifically focused on using embedding models effectively:

```
def print_embedding_best_practices():
    """Print best practices for using embedding models in search"""

    print("\n=== BEST PRACTICES FOR EMBEDDING MODELS IN HYBRID SEARCH
===\n")

    best_practices = [
        {
            "area": "Embedding Model Selection",
            "practices": [
                "Choose a model appropriate for your domain (general vs.
specialized)",
                "Consider the dimensionality of embeddings (higher
dimensions = more storage & processing costs)",
                "Evaluate different models on your specific data before
deployment",
                "Consider using models that support multiple languages if
needed"
            ]
        },
        {
            "area": "Vector Index Configuration",
            "practices": [
                "Adjust HNSW parameters (m, efConstruction, efSearch)
based on dataset size and search requirements",
                "Use 'cosine' similarity for most text search
applications",
                "Consider using 'euclidean' for geographic or visual
search applications",
                "Optimize for either speed (lower m, ef values) or
accuracy (higher values) based on needs"
            ]
        },
        {
            "area": "Hybrid Search Implementation",
            "practices": [
                "Balance weights between vector and keyword components
based on testing",
                "Use vector search for semantic understanding and keyword
search for precision",
                "Add filters on metadata to narrow results while
preserving semantic relevance",
                "Use semantic configurations for further re-ranking when
available"
            ]
        },
        {
            "area": "Performance Optimization",
            "practices": [
                "Generate embeddings at indexing time rather than query
time when possible",
```

```
                "Consider chunking large documents into smaller segments
for more precise vector matching",
                "Cache query embeddings for frequently used queries",
                "Monitor vector index size and search latency as your data
grows"
            ]
        }
    ]

    for area in best_practices:
        print(f"\n{area['area']}:")
        for i, practice in enumerate(area['practices'], 1):
            print(f"  {i}. {practice}")
```

This comprehensive guide should now properly address the critical embedding model configuration needed for both semantic and hybrid search in Azure AI Search. The examples cover:

1. Configuring vector fields in your index schema
2. Setting up the HNSW algorithm for vector search
3. Generating embeddings using OpenAI (or Azure OpenAI)
4. Uploading documents with their embeddings
5. Performing different types of searches:
    - Pure vector search
    - Hybrid (text + vector) search
    - Semantic hybrid search with reranking

By properly configuring embedding models for both indexing and query vectorization, you can create powerful search experiences that combine the best of traditional keyword search, vector similarity search, and semantic understanding.