# Q

Is there any way to upload complete documents and ask Azure to break the document in chunks and vectorize it ?

# A

Yes, Azure AI Search provides capabilities to upload complete documents, break them into chunks, and vectorize them for more effective searching. This is typically handled through Azure AI Search's skillsets and indexers, which can be part of an AI enrichment pipeline. Let me guide you through the process.

## Uploading Complete Documents, Chunking, and Vectorizing with Azure AI Search

### 1. Setting Up a Document Processing Pipeline

First, you'll need to set up a pipeline that can:

1. Ingest complete documents (PDF, Word, text, etc.)
2. Extract text from these documents
3. Split the text into manageable chunks
4. Generate vector embeddings for each chunk
5. Index both the chunks and their vectors

Here's how to implement this in Python:

```
from azure.identity import AzureCliCredential
from azure.search.documents.indexes import SearchIndexClient
from azure.search.documents.indexes.models import (
    SearchIndex,
    SearchField,
    SearchFieldDataType,
    SimpleField,
    SearchableField,
    VectorSearch,
    VectorSearchProfile,
    VectorSearchAlgorithmConfiguration,
    HnswAlgorithmConfiguration
)
from azure.search.documents import SearchClient
from azure.core.credentials import AzureKeyCredential
from azure.storage.blob import BlobServiceClient, BlobClient
from azure.ai.formrecognizer import DocumentAnalysisClient
from azure.mgmt.search import SearchManagementClient
from azure.mgmt.search.models import SearchService, Sku
import openai
import time
import uuid
```

```
def setup_document_processing_pipeline(
        resource_group_name,
        storage_account_name,
        storage_account_key,
        search_service_name,
        search_admin_key,
        form_recognizer_endpoint,
        form_recognizer_key,
        openai_api_key=None,
        azure_openai_endpoint=None,
        azure_openai_key=None,
        azure_openai_deployment=None):
    """
    Setup a complete pipeline for document processing, chunking and
vectorization
    """
    # 1. Setup storage for documents
    blob_service_client = BlobServiceClient(

account_url=f"https://{storage_account_name}.blob.core.windows.net",
        credential=storage_account_key
    )

    # Create a container for documents if it doesn't exist
    container_name = "documents"
    try:
        blob_service_client.create_container(container_name)
        print(f"Container '{container_name}' created successfully.")
    except Exception as e:
        if "ContainerAlreadyExists" in str(e):
            print(f"Container '{container_name}' already exists.")
        else:
            print(f"Error creating container: {str(e)}")

    # 2. Setup search client
    search_endpoint = f"https://{search_service_name}.search.windows.net"
    search_credential = AzureKeyCredential(search_admin_key)
    index_client = SearchIndexClient(endpoint=search_endpoint,
credential=search_credential)

    # 3. Setup Form Recognizer client for document text extraction
    form_recognizer_client = DocumentAnalysisClient(
        endpoint=form_recognizer_endpoint,
        credential=AzureKeyCredential(form_recognizer_key)
    )

    # 4. Setup OpenAI for embeddings
    if openai_api_key:
        # Use public OpenAI API
        openai.api_key = openai_api_key
        embedding_source = "openai"
    elif azure_openai_endpoint and azure_openai_key:
        # Use Azure OpenAI
```

```
        openai.api_type = "azure"
        openai.api_base = azure_openai_endpoint
        openai.api_key = azure_openai_key
        openai.api_version = "2023-05-15"
        embedding_source = "azure_openai"
    else:
        print("Warning: No embedding service configured. Vectorization
will not be available.")
        embedding_source = None

    return {
        "blob_service_client": blob_service_client,
        "container_name": container_name,
        "search_endpoint": search_endpoint,
        "search_credential": search_credential,
        "index_client": index_client,
        "form_recognizer_client": form_recognizer_client,
        "embedding_source": embedding_source,
        "azure_openai_deployment": azure_openai_deployment
    }
```

## 2. Creating an Index for Chunked Documents

Next, we need to create an index designed to store document chunks with their embeddings:

```
def create_chunked_document_index(index_client, index_name="chunked-
documents"):
    """
    Create an index specifically designed for chunked documents with
vector embeddings
    """
    # Define fields for the chunked document index
    fields = [
        # Document and chunk identifiers
        SimpleField(name="id", type=SearchFieldDataType.String, key=True),
        SimpleField(name="document_id", type=SearchFieldDataType.String,
filterable=True, facetable=True),
        SimpleField(name="chunk_id", type=SearchFieldDataType.Int32,
filterable=True, sortable=True),

        # Original document metadata
        SimpleField(name="document_name", type=SearchFieldDataType.String,
filterable=True, facetable=True),
        SimpleField(name="document_url", type=SearchFieldDataType.String),
        SimpleField(name="file_type", type=SearchFieldDataType.String,
filterable=True, facetable=True),
        SimpleField(name="created_date",
type=SearchFieldDataType.DateTimeOffset, filterable=True, sortable=True),
        SimpleField(name="last_modified",
type=SearchFieldDataType.DateTimeOffset, filterable=True, sortable=True),
```

```
        # Chunk content and metadata
        SearchableField(name="chunk_text",
type=SearchFieldDataType.String, analyzer_name="en.microsoft"),
        SimpleField(name="chunk_size", type=SearchFieldDataType.Int32),
        SimpleField(name="page_number", type=SearchFieldDataType.Int32,
filterable=True),

        # Vector embeddings
        SimpleField(
            name="chunk_embedding",

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

        # Additional metadata
        SimpleField(name="source", type=SearchFieldDataType.String,
filterable=True, facetable=True),
        SimpleField(name="tags",
type=SearchFieldDataType.Collection(SearchFieldDataType.String),
                    filterable=True, facetable=True)
    ]

    # Configure vector search
    vector_search = VectorSearch(
        algorithms=[
            VectorSearchAlgorithmConfiguration(
                name="default-algorithm",
                kind="hnsw",
                hnsw_parameters=HnswAlgorithmConfiguration(
                    m=4,
                    ef_construction=400,
                    ef_search=500,
                    metric="cosine"
                )
            )
        ],
        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"Chunked document index '{index_name}' created
successfully.")
        return result
    except Exception as e:
        print(f"Error creating chunked document index: {str(e)}")
        raise
```

### 3. Uploading and Processing Documents

Now, let's create functions to upload documents, extract text, chunk it, and vectorize each chunk:

```
def upload_document_to_blob(blob_service_client, container_name,
local_file_path, blob_name=None):
    """
    Upload a document to Azure Blob Storage
    """
    if blob_name is None:
        # Use the local filename if no blob name is provided
        import os
        blob_name = os.path.basename(local_file_path)

    blob_client =
blob_service_client.get_blob_client(container=container_name,
blob=blob_name)

    with open(local_file_path, "rb") as data:
        blob_client.upload_blob(data, overwrite=True)

    print(f"Uploaded {local_file_path} to {container_name}/{blob_name}")
    return
f"https://{blob_service_client.account_name}.blob.core.windows.net/{contai
ner_name}/{blob_name}"

def extract_text_from_document(form_recognizer_client, document_url):
    """
    Extract text from a document using Form Recognizer
    """
    try:
        poller =
form_recognizer_client.begin_analyze_document_from_url("prebuilt-
document", document_url)
        result = poller.result()

        # Extract text content, maintaining page structure
        pages = []
        for page in result.pages:
            page_text = ""
            for line in page.lines:
                page_text += line.content + "\n"
            pages.append({
```

```
                "page_number": page.page_number,
                "text": page_text
            })

        return pages
    except Exception as e:
        print(f"Error extracting text from document: {str(e)}")
        raise

def chunk_document(pages, chunk_size=1000, overlap=100):
    """
    Split document text into chunks with optional overlap
    """
    chunks = []

    for page in pages:
        page_text = page["text"]
        page_number = page["page_number"]

        # If the page text is shorter than the chunk size, keep it as one
chunk
        if len(page_text) <= chunk_size:
            chunks.append({
                "page_number": page_number,
                "chunk_id": len(chunks),
                "text": page_text,
                "size": len(page_text)
            })
            continue

        # Split the page into overlapping chunks
        start = 0
        while start < len(page_text):
            # Calculate the end position of this chunk
            end = min(start + chunk_size, len(page_text))

            # If we're not at the end of the text, try to break at a
sentence or paragraph
            if end < len(page_text):
                # Look for paragraph break
                paragraph_break = page_text.rfind('\n\n', start, end)
                if paragraph_break != -1 and paragraph_break > start +
chunk_size // 2:
                    end = paragraph_break + 2  # Include the newlines
                else:
                    # Look for sentence break (period followed by space or
newline)
                    sentence_break = max(
                        page_text.rfind('. ', start, end),
                        page_text.rfind('.\n', start, end)
                    )
                    if sentence_break != -1 and sentence_break > start +
chunk_size // 2:
                        end = sentence_break + 2  # Include the period and
```

```
space/newline

            chunk_text = page_text[start:end].strip()
            if chunk_text:  # Only add non-empty chunks
                chunks.append({
                    "page_number": page_number,
                    "chunk_id": len(chunks),
                    "text": chunk_text,
                    "size": len(chunk_text)
                })

            # Move to the next chunk, with overlap
            start = end - overlap if end < len(page_text) else
len(page_text)

    return chunks

def generate_embedding(text, embedding_source,
azure_openai_deployment=None):
    """
    Generate vector embedding for text using the configured embedding
source
    """
    if embedding_source == "openai":
        response = openai.embeddings.create(
            model="text-embedding-ada-002",
            input=text
        )
        return response.data[0].embedding

    elif embedding_source == "azure_openai":
        response = openai.embeddings.create(
            input=text,
            deployment_id=azure_openai_deployment
        )
        return response.data[0].embedding

    else:
        print("No embedding source configured. Returning empty
embedding.")
        return []

def process_and_index_document(
        local_file_path,
        pipeline_config,
        index_name="chunked-documents",
        chunk_size=1000,
        overlap=100,
        tags=None):
    """
    Complete process to upload, extract, chunk, vectorize and index a
document
    """
    # Extract components from pipeline config
```

```
    blob_service_client = pipeline_config["blob_service_client"]
    container_name = pipeline_config["container_name"]
    form_recognizer_client = pipeline_config["form_recognizer_client"]
    index_client = pipeline_config["index_client"]
    search_endpoint = pipeline_config["search_endpoint"]
    search_credential = pipeline_config["search_credential"]
    embedding_source = pipeline_config["embedding_source"]
    azure_openai_deployment = pipeline_config["azure_openai_deployment"]

    # 1. Upload document to blob storage
    import os
    document_name = os.path.basename(local_file_path)
    file_type = os.path.splitext(document_name)[1].lower().replace('.',
'')

    document_url = upload_document_to_blob(
        blob_service_client,
        container_name,
        local_file_path
    )

    # Generate a unique document ID
    document_id = str(uuid.uuid4())

    # 2. Extract text using Form Recognizer
    print(f"Extracting text from {document_name}...")
    pages = extract_text_from_document(form_recognizer_client,
document_url)

    # 3. Chunk the document
    print(f"Chunking document into segments (size: {chunk_size}, overlap:
{overlap})...")
    chunks = chunk_document(pages, chunk_size, overlap)
    print(f"Created {len(chunks)} chunks from the document.")

    # 4. Generate embeddings and prepare documents for indexing
    print("Generating embeddings and preparing for indexing...")
    search_documents = []

    for chunk in chunks:
        # Generate embedding for this chunk
        embedding = generate_embedding(
            chunk["text"],
            embedding_source,
            azure_openai_deployment
        )

        # Create a search document for this chunk
        search_document = {
            "id": f"{document_id}-{chunk['chunk_id']}",
            "document_id": document_id,
            "chunk_id": chunk["chunk_id"],
            "document_name": document_name,
            "document_url": document_url,
```

```
            "file_type": file_type,
            "created_date": time.strftime("%Y-%m-%dT%H:%M:%SZ",
time.gmtime()),
            "last_modified": time.strftime("%Y-%m-%dT%H:%M:%SZ",
time.gmtime()),
            "chunk_text": chunk["text"],
            "chunk_size": chunk["size"],
            "page_number": chunk["page_number"],
            "chunk_embedding": embedding,
            "source": "document_upload"
        }

        # Add tags if provided
        if tags:
            search_document["tags"] = tags

        search_documents.append(search_document)

    # 5. Index the chunks
    search_client = SearchClient(
        endpoint=search_endpoint,
        index_name=index_name,
        credential=search_credential
    )

    # Split into batches of 1000 documents (Azure Search limit)
    batch_size = 1000
    for i in range(0, len(search_documents), batch_size):
        batch = search_documents[i:i + batch_size]
        result = search_client.upload_documents(documents=batch)
        print(f"Indexed batch {i // batch_size +
1}/{(len(search_documents) + batch_size - 1) // batch_size}: {len(result)}
chunks")

    print(f"Document processing complete. Indexed {len(search_documents)}
chunks from '{document_name}'.")
    return document_id
```

## 4. Querying the Chunked Documents

Now, let's create functions to query the chunked documents:

```
def semantic_search_document_chunks(
        search_client,
        query_text,
        embedding_source,
        azure_openai_deployment=None,
        document_id=None,
        top=10):
    """
    Perform a semantic search on document chunks
```

```
    """
    # Generate embedding for the query
    query_embedding = generate_embedding(
        query_text,
        embedding_source,
        azure_openai_deployment
    )

    # Build search options
    search_options = {
        "search_text": query_text,  # For keyword search
        "vector": query_embedding,   # For vector search
        "vector_fields": ["chunk_embedding"],  # Field to perform vector
search on
        "select":
"id,document_name,document_url,chunk_text,page_number,chunk_id",
        "top": top,
        "include_total_count": True,
        "highlight_fields": "chunk_text",
        "highlight_pre_tag": "<b>",
        "highlight_post_tag": "</b>"
    }

    # Add filter for specific document if provided
    if document_id:
        search_options["filter"] = f"document_id eq '{document_id}'"

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

        # Process and return the results
        processed_results = []
        for result in results:
            processed_result = {
                "id": result["id"],
                "document_name": result["document_name"],
                "document_url": result["document_url"],
                "page_number": result["page_number"],
                "chunk_id": result["chunk_id"],
                "score": result["@search.score"],
                "text": result["chunk_text"]
            }

            # Add highlights if available
            if '@search.highlights' in result and 'chunk_text' in
result['@search.highlights']:
                processed_result["highlights"] =
result['@search.highlights']['chunk_text']

            processed_results.append(processed_result)

        return processed_results
```

```
        except Exception as e:
            print(f"Search error: {str(e)}")
            return []
```

## 5. Complete Example: Process and Query a Document

Here's a complete example that puts everything together:

```
def main():
    """Main function to demonstrate document processing and chunking"""
    # Get configuration parameters from user
    resource_group_name = input("Enter your resource group name: ")
    storage_account_name = input("Enter your storage account name: ")
    storage_account_key = input("Enter your storage account key: ")
    search_service_name = input("Enter your search service name: ")
    search_admin_key = input("Enter your search admin key: ")
    form_recognizer_endpoint = input("Enter your Form Recognizer endpoint:
")
    form_recognizer_key = input("Enter your Form Recognizer key: ")

    # Choose embedding service
    embedding_choice = input("Use OpenAI (1) or Azure OpenAI (2) for
embeddings? ")

    if embedding_choice == "1":
        openai_api_key = input("Enter your OpenAI API key: ")
        azure_openai_endpoint = None
        azure_openai_key = None
        azure_openai_deployment = None
    else:
        openai_api_key = None
        azure_openai_endpoint = input("Enter your Azure OpenAI endpoint:
")
        azure_openai_key = input("Enter your Azure OpenAI key: ")
        azure_openai_deployment = input("Enter your Azure OpenAI
embeddings deployment name: ")

    # Set up the pipeline
    print("\nSetting up document processing pipeline...")
    pipeline_config = setup_document_processing_pipeline(
        resource_group_name,
        storage_account_name,
        storage_account_key,
        search_service_name,
        search_admin_key,
        form_recognizer_endpoint,
        form_recognizer_key,
        openai_api_key,
        azure_openai_endpoint,
        azure_openai_key,
        azure_openai_deployment
```

```
    )

    # Create the index
    index_name = "chunked-documents"
    print(f"\nCreating index '{index_name}'...")
    create_chunked_document_index(pipeline_config["index_client"],
index_name)

    # Process a document
    document_path = input("\nEnter the path to a document to process: ")

    # Get optional tags
    tags_input = input("Enter optional tags (comma-separated) or press
Enter for none: ")
    tags = [tag.strip() for tag in tags_input.split(',')] if
tags_input.strip() else None

    # Process the document
    print("\nProcessing document...")
    document_id = process_and_index_document(
        document_path,
        pipeline_config,
        index_name=index_name,
        tags=tags
    )
    print(f"Document processed with ID: {document_id}")

    # Create a search client
    search_client = SearchClient(
        endpoint=pipeline_config["search_endpoint"],
        index_name=index_name,
        credential=pipeline_config["search_credential"]
    )

    # Interactive search mode
    print("\n=== INTERACTIVE DOCUMENT SEARCH ===")
    print("Enter your queries to search the document chunks. Type 'exit'
to quit.")

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

        # Search for chunks
        results = semantic_search_document_chunks(
            search_client,
            query,
            pipeline_config["embedding_source"],
            pipeline_config["azure_openai_deployment"],
            document_id  # Optional: limit to the current document
        )

        # Display results
```

```
        print(f"\nFound {len(results)} relevant chunks:")
        for i, result in enumerate(results, 1):
            print(f"\n{i}. Document: {result['document_name']} (Page
{result['page_number']})")
            print(f"   Score: {result['score']:.4f}")

            # Show highlights if available
            if "highlights" in result:
                print("   Highlights:")
                for highlight in result["highlights"]:
                    print(f"   ...{highlight}...")
            else:
                # Show a text snippet
                text = result["text"]
                snippet = text[:200] + "..." if len(text) > 200 else text
                print(f"   Text: {snippet}")

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

## 6. Using Azure Cognitive Search's Built-in Chunking and Vectorization

Azure AI Search also provides built-in capabilities for document chunking and vectorization through skillsets and indexers. This approach is more declarative and less code-intensive:

```
from azure.core.credentials import AzureKeyCredential
from azure.search.documents.indexes import SearchIndexClient
from azure.search.documents.indexes.models import (
    SearchIndex,
    SearchField,
    SearchFieldDataType,
    SimpleField,
    SearchableField,
    IndexingParameters,
    SearchIndexer,
    SearchIndexerDataSourceConnection,
    SearchIndexerDataContainer,
    SearchIndexerSkillset,
    OcrSkill,
    TextSplitSkill,
    DocumentExtractionSkill,
    EntityRecognitionSkill,
    SplitSkill
)

def setup_indexer_pipeline(
        resource_group_name,
        storage_connection_string,
        search_service_name,
        search_admin_key,
        cognitive_services_key):
```

```
    """
    Setup a pipeline using Azure Search's built-in indexers, skillsets for
chunking
    """
    search_endpoint = f"https://{search_service_name}.search.windows.net"
    search_credential = AzureKeyCredential(search_admin_key)

    # Create the client
    index_client = SearchIndexClient(endpoint=search_endpoint,
credential=search_credential)

    # 1. Create an index
    index_name = "auto-chunked-docs"
    index = SearchIndex(
        name=index_name,
        fields=[
            SimpleField(name="id", type=SearchFieldDataType.String,
key=True),
            SearchableField(name="content",
type=SearchFieldDataType.String, analyzer_name="en.microsoft"),
            SimpleField(name="metadata_storage_name",
type=SearchFieldDataType.String, filterable=True, facetable=True),
            SimpleField(name="metadata_storage_path",
type=SearchFieldDataType.String, filterable=True),
            SimpleField(name="metadata_content_type",
type=SearchFieldDataType.String, filterable=True, facetable=True),
            SimpleField(name="chunk_id", type=SearchFieldDataType.String,
filterable=True),
            SearchableField(name="chunk_content",
type=SearchFieldDataType.String, analyzer_name="en.microsoft"),
            SimpleField(name="page_number",
type=SearchFieldDataType.Int32, filterable=True),
            SimpleField(name="chunk_embedding",
type=SearchFieldDataType.Collection(SearchFieldDataType.Single),
                        vector_search_dimensions=1536,
vector_search_profile_name="default-profile")
        ]
    )

    index_client.create_or_update_index(index)
    print(f"Index '{index_name}' created.")

    # 2. Create a data source (requires a connection string to blob
storage)
    from azure.search.documents.indexes import SearchIndexerClient
    indexer_client = SearchIndexerClient(endpoint=search_endpoint,
credential=search_credential)

    data_source_name = "document-blob-data"
    data_source = SearchIndexerDataSourceConnection(
        name=data_source_name,
        type="azureblob",
        connection_string=storage_connection_string,
        container=SearchIndexerDataContainer(name="documents")
```

```
    )

    indexer_client.create_or_update_data_source_connection(data_source)
    print(f"Data source '{data_source_name}' created.")

    # 3. Create a skillset with document extraction, chunking, and
vectorization skills
    skillset_name = "document-chunking-skillset"

    # This is a simplified example - actual skillset would be more complex
    skillset = SearchIndexerSkillset(
        name=skillset_name,
        description="Skillset for document extraction, chunking, and
vectorization",
        skills=[
            # Extract content from various document formats
            DocumentExtractionSkill(
                inputs=[
                    {"name": "content", "source": "/document/content"},
                    {"name": "contentType", "source":
"/document/contentType"}
                ],
                outputs=[
                    {"name": "extractedContent", "targetName":
"extractedContent"}
                ],
                parameters={"parsingMode": "default"}
            ),

            # Split text into chunks
            SplitSkill(
                inputs=[
                    {"name": "text", "source":
"/document/extractedContent"},
                ],
                outputs=[
                    {"name": "textItems", "targetName": "textItems"}
                ],
                parameters={
                    "textSplitMode": "pages",  # Can be "pages",
"sentences" or "lines"
                    "maximumPageLength": 5000, # Characters per chunk
                    "defaultLanguageCode": "en"
                }
            ),

            # Here you would typically add a custom skill to generate
embeddings
            # Azure AI Search now supports vectorization through custom
skills or Azure OpenAI integration
        ],
        cognitive_services_account={"key": cognitive_services_key}
    )
```

```
    indexer_client.create_or_update_skillset(skillset)
    print(f"Skillset '{skillset_name}' created.")

    # 4. Create an indexer
    indexer_name = "document-chunking-indexer"
    indexer = SearchIndexer(
        name=indexer_name,
        description="Indexer for chunked documents",
        data_source_name=data_source_name,
        target_index_name=index_name,
        skillset_name=skillset_name,
        parameters=IndexingParameters(
            batch_size=1,
            max_failed_items=0,
            max_failed_items_per_batch=0
        ),
        field_mappings=[
            # Base mappings for the document
            {"sourceFieldName": "metadata_storage_name",
"targetFieldName": "metadata_storage_name"},
            {"sourceFieldName": "metadata_storage_path",
"targetFieldName": "metadata_storage_path"},
            {"sourceFieldName": "metadata_content_type",
"targetFieldName": "metadata_content_type"},
        ],
        output_field_mappings=[
            # Map chunked output to fields
            {"sourceFieldName": "/document/textItems/*",
"targetFieldName": "chunk_content"},
            {"sourceFieldName": "/document/textItems/*/page_id",
"targetFieldName": "chunk_id"},
            {"sourceFieldName": "/document/textItems/*/page_number",
"targetFieldName": "page_number"}
            # In a complete example, you would also map the embeddings to
the chunk_embedding field
        ]
    )

    indexer_client.create_or_update_indexer(indexer)
    print(f"Indexer '{indexer_name}' created.")

    # 5. Run the indexer to start processing documents
    indexer_client.run_indexer(indexer_name)
    print(f"Indexer '{indexer_name}' started.")

# Note: This declarative approach requires appropriate settings in Azure
Portal
# Some advanced features like vector embedding generation might require
custom skills
```

## 7. Azure AI Search REST API Approach (Declarative)

For the most complete and declarative approach, you can use the Azure AI Search REST API to define your entire pipeline. Here's how it would look (pseudocode):

```
# This would be implemented using requests library to make direct REST API
calls

# 1. Create the index with vector search capabilities
index_definition = {
    "name": "chunked-documents",
    "fields": [
        {"name": "id", "type": "Edm.String", "key": True, "searchable":
False},
        {"name": "content", "type": "Edm.String", "searchable": True,
"analyzer": "en.microsoft"},
        {"name": "chunk_id", "type": "Edm.String", "filterable": True},
        {"name": "chunk_text", "type": "Edm.String", "searchable": True,
"analyzer": "en.microsoft"},
        {
            "name": "chunk_vector",
            "type": "Collection(Edm.Single)",
            "searchable": False,
            "dimensions": 1536,
            "vectorSearchProfile": "my-profile"
        }
        # Additional fields...
    ],
    "vectorSearch": {
        "algorithms": [
            {
                "name": "my-algorithm",
                "kind": "hnsw",
                "hnswParameters": {
                    "m": 4,
                    "efConstruction": 400,
                    "efSearch": 500,
                    "metric": "cosine"
                }
            }
        ],
        "profiles": [
            {
                "name": "my-profile",
                "algorithmConfigurationName": "my-algorithm"
            }
        ]
    }
}

# 2. Create a skillset with chunking and vectorization (continued)
skillset_definition = {
    "name": "chunking-vectorization-skillset",
    "description": "Skillset for document chunking and vectorization",
```

```
    "skills": [
        # Document extraction skill
        {
            "@odata.type":
"#Microsoft.Skills.Text.DocumentExtractionSkill",
            "context": "/document",
            "inputs": [
                { "name": "content", "source": "/document/content" },
                { "name": "contentType", "source": "/document/contentType"
}
            ],
            "outputs": [
                { "name": "text", "targetName": "extractedText" },
                { "name": "pages", "targetName": "pages" }
            ],
            "parameters": { "parsingMode": "default" }
        },

        # Text split skill
        {
            "@odata.type": "#Microsoft.Skills.Text.SplitSkill",
            "context": "/document",
            "textSplitMode": "pages",
            "maximumPageLength": 5000,
            "inputs": [
                { "name": "text", "source": "/document/extractedText" }
            ],
            "outputs": [
                { "name": "textItems", "targetName": "chunks" }
            ]
        },

        # Vector embedding generation using Azure OpenAI
        {
            "@odata.type":
"#Microsoft.Skills.Text.AzureOpenAIEmbeddingSkill",
            "context": "/document/chunks/*",
            "inputs": [
                { "name": "text", "source": "/document/chunks/*" }
            ],
            "outputs": [
                { "name": "embedding", "targetName": "vector" }
            ],
            "resourceUri": "https://your-openai-
resource.openai.azure.com/",
            "deploymentId": "your-embedding-deployment-name",
            "apiKey": "your-openai-api-key"
        }
    ],
    "cognitiveServices": {
        "@odata.type": "#Microsoft.Azure.Search.CognitiveServicesByKey",
        "description": "Cognitive Services",
        "key": "your-cognitive-services-key"
    }
```

```
}

# 3. Create a data source pointing to your document storage
datasource_definition = {
    "name": "my-documents",
    "type": "azureblob",
    "credentials": {
        "connectionString": "your-storage-connection-string"
    },
    "container": {
        "name": "documents"
    }
}

# 4. Create an indexer that uses the skillset
indexer_definition = {
    "name": "document-indexer",
    "dataSourceName": "my-documents",
    "targetIndexName": "chunked-documents",
    "skillsetName": "chunking-vectorization-skillset",
    "parameters": {
        "configuration": {
            "dataToExtract": "contentAndMetadata",
            "parsingMode": "default"
        }
    },
    "fieldMappings": [
        { "sourceFieldName": "metadata_storage_name", "targetFieldName":
"documentName" },
        { "sourceFieldName": "metadata_storage_path", "targetFieldName":
"documentUrl" },
        { "sourceFieldName": "metadata_content_type", "targetFieldName":
"contentType" }
    ],
    "outputFieldMappings": [
        { "sourceFieldName": "/document/chunks/*", "targetFieldName":
"chunk_text" },
        { "sourceFieldName": "/document/chunks/*/vector",
"targetFieldName": "chunk_vector" }
    ]
}
```

## 8. Putting It All Together: Complete Solution

Here's a complete application that implements document uploading, chunking, and vectorization with Azure AI Search:

```
import os
import asyncio
import uuid
from azure.identity import DefaultAzureCredential
```

```
from azure.storage.blob import BlobServiceClient
from azure.core.credentials import AzureKeyCredential
from azure.search.documents.indexes import SearchIndexClient
from azure.search.documents import SearchClient
from azure.ai.formrecognizer import DocumentAnalysisClient
from azure.search.documents.indexes.models import (
    SearchIndex, SearchField, SearchFieldDataType, SimpleField,
SearchableField,
    VectorSearch, VectorSearchProfile, VectorSearchAlgorithmConfiguration,
HnswAlgorithmConfiguration
)
import openai

class DocumentProcessor:
    def __init__(self, config):
        self.config = config
        # Initialize clients
        self.blob_service_client =
BlobServiceClient.from_connection_string(config["storage_connection_string
"])
        self.container_client =
self.blob_service_client.get_container_client(config["container_name"])
        self.search_credential =
AzureKeyCredential(config["search_api_key"])
        self.search_index_client = SearchIndexClient(

endpoint=f"https://{config['search_service_name']}.search.windows.net",
            credential=self.search_credential
        )
        self.search_client = SearchClient(

endpoint=f"https://{config['search_service_name']}.search.windows.net",
            index_name=config["index_name"],
            credential=self.search_credential
        )
        self.form_recognizer_client = DocumentAnalysisClient(
            endpoint=config["form_recognizer_endpoint"],
            credential=AzureKeyCredential(config["form_recognizer_key"])
        )

        # Setup OpenAI
        openai.api_type = "azure"
        openai.api_base = config["openai_endpoint"]
        openai.api_key = config["openai_key"]
        openai.api_version = "2023-05-15"

    async def setup(self):
        """Set up the required Azure resources"""
        # Create container if it doesn't exist
        try:
            self.container_client.create_container()
            print(f"Container '{self.config['container_name']}' created.")
        except Exception as e:
            if "ContainerAlreadyExists" in str(e):
```

```
                print(f"Container '{self.config['container_name']}'
already exists.")
            else:
                print(f"Error creating container: {str(e)}")

        # Create index if it doesn't exist
        try:
            index = self._create_index_definition()
            self.search_index_client.create_or_update_index(index)
            print(f"Index '{self.config['index_name']}' created or
updated.")
        except Exception as e:
            print(f"Error creating index: {str(e)}")
            raise

    def _create_index_definition(self):
        """Create the search index definition"""
        fields = [
            SimpleField(name="id", type=SearchFieldDataType.String,
key=True),
            SimpleField(name="document_id",
type=SearchFieldDataType.String, filterable=True),
            SimpleField(name="document_name",
type=SearchFieldDataType.String, filterable=True, facetable=True),
            SimpleField(name="document_url",
type=SearchFieldDataType.String),
            SimpleField(name="chunk_id", type=SearchFieldDataType.Int32,
filterable=True, sortable=True),
            SimpleField(name="page_number",
type=SearchFieldDataType.Int32, filterable=True),
            SearchableField(name="chunk_text",
type=SearchFieldDataType.String, analyzer_name="en.microsoft"),
            SimpleField(
                name="vector",

type=SearchFieldDataType.Collection(SearchFieldDataType.Single),
                vector_search_dimensions=1536,
                vector_search_profile_name="vector-profile"
            )
        ]

        vector_search = VectorSearch(
            algorithms=[
                VectorSearchAlgorithmConfiguration(
                    name="vector-algorithm",
                    kind="hnsw",
                    hnsw_parameters=HnswAlgorithmConfiguration(
                        m=4,
                        ef_construction=400,
                        ef_search=500,
                        metric="cosine"
                    )
                )
            ],
```

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

        return SearchIndex(
            name=self.config["index_name"],
            fields=fields,
            vector_search=vector_search
        )

    async def process_document(self, file_path):
        """Process a document: upload, extract text, chunk, vectorize, and
index"""
        document_id = str(uuid.uuid4())
        document_name = os.path.basename(file_path)

        # 1. Upload to blob storage
        print(f"Uploading {document_name} to blob storage...")
        blob_client = self.container_client.get_blob_client(document_name)
        with open(file_path, "rb") as data:
            blob_client.upload_blob(data, overwrite=True)

        document_url = blob_client.url

        # 2. Extract text using Form Recognizer
        print("Extracting text with Form Recognizer...")
        poller =
self.form_recognizer_client.begin_analyze_document_from_url(
            "prebuilt-document", document_url
        )
        result = poller.result()

        # 3. Chunk the text
        chunks = []
        chunk_id = 0

        for page in result.pages:
            page_text = ""
            for line in page.lines:
                page_text += line.content + "\n"

            # Break long pages into smaller chunks
            chunk_size = 4000  # Characters per chunk
            overlap = 200      # Characters of overlap between chunks

            if len(page_text) <= chunk_size:
                chunks.append({
                    "page_number": page.page_number,
                    "chunk_id": chunk_id,
                    "text": page_text
```

```
                })
                chunk_id += 1
            else:
                # Split into overlapping chunks
                start = 0
                while start < len(page_text):
                    end = min(start + chunk_size, len(page_text))

                    # Try to break at paragraph or sentence
                    if end < len(page_text):
                        paragraph_break = page_text.rfind("\n\n", start,
end)
                        if paragraph_break > start + chunk_size // 2:
                            end = paragraph_break + 2
                        else:
                            sentence_break = max(
                                page_text.rfind(". ", start, end),
                                page_text.rfind(".\n", start, end)
                            )
                            if sentence_break > start + chunk_size // 2:
                                end = sentence_break + 2

                    chunk_text = page_text[start:end].strip()
                    if chunk_text:
                        chunks.append({
                            "page_number": page.page_number,
                            "chunk_id": chunk_id,
                            "text": chunk_text
                        })
                        chunk_id += 1

                    start = end - overlap if end < len(page_text) else
len(page_text)

        print(f"Created {len(chunks)} chunks from document.")

        # 4. Vectorize and index chunks
        search_documents = []

        for chunk in chunks:
            # Generate embedding
            embedding_response = openai.embeddings.create(
                input=chunk["text"],
                deployment_id=self.config["openai_embedding_deployment"]
            )
            vector = embedding_response.data[0].embedding

            # Create search document
            search_document = {
                "id": f"{document_id}-{chunk['chunk_id']}",
                "document_id": document_id,
                "document_name": document_name,
                "document_url": document_url,
                "chunk_id": chunk["chunk_id"],
```

```
                "page_number": chunk["page_number"],
                "chunk_text": chunk["text"],
                "vector": vector
            }

            search_documents.append(search_document)

        # 5. Index in batches
        batch_size = 100  # Azure Search has limits on batch size
        for i in range(0, len(search_documents), batch_size):
            batch = search_documents[i:i + batch_size]
            result = self.search_client.upload_documents(documents=batch)
            print(f"Indexed batch {i//batch_size +
1}/{(len(search_documents) + batch_size - 1)//batch_size}")

        print(f"Document processing complete. Document ID: {document_id}")
        return document_id

    async def search(self, query_text, document_id=None, top=5):
        """Search for chunks using both keyword and vector search"""
        # Generate query vector
        embedding_response = openai.embeddings.create(
            input=query_text,
            deployment_id=self.config["openai_embedding_deployment"]
        )
        vector = embedding_response.data[0].embedding

        # Set up search options
        options = {
            "search_text": query_text,
            "vector": vector,
            "vector_fields": "vector",
            "top": top,
            "select":
"id,document_name,document_url,chunk_id,page_number,chunk_text",
            "include_total_count": True,
            "highlight_fields": "chunk_text",
            "highlight_pre_tag": "<em>",
            "highlight_post_tag": "</em>"
        }

        # Add filter for specific document
        if document_id:
            options["filter"] = f"document_id eq '{document_id}'"

        # Execute search
        results = self.search_client.search(**options)

        # Process results
        processed_results = []
        for result in results:
            processed_result = {
                "id": result["id"],
                "document_name": result["document_name"],
```

```
                "document_url": result["document_url"],
                "chunk_id": result["chunk_id"],
                "page_number": result["page_number"],
                "score": result["@search.score"],
                "text": result["chunk_text"]
            }

            if "@search.highlights" in result and "chunk_text" in
result["@search.highlights"]:
                processed_result["highlights"] =
result["@search.highlights"]["chunk_text"]

            processed_results.append(processed_result)

        return processed_results

# Example usage
async def main():
    config = {
        "storage_connection_string": "your_connection_string",
        "container_name": "documents",
        "search_service_name": "your-search-service",
        "search_api_key": "your-search-api-key",
        "index_name": "document-chunks",
        "form_recognizer_endpoint": "your-form-recognizer-endpoint",
        "form_recognizer_key": "your-form-recognizer-key",
        "openai_endpoint": "your-azure-openai-endpoint",
        "openai_key": "your-azure-openai-key",
        "openai_embedding_deployment": "your-embedding-deployment-name"
    }

    processor = DocumentProcessor(config)
    await processor.setup()

    # Process a document
    document_path = "path/to/your/document.pdf"
    document_id = await processor.process_document(document_path)

    # Search within the document
    query = "What is the main topic of this document?"
    results = await processor.search(query, document_id=document_id)

    print("\nSearch Results:")
    for i, result in enumerate(results, 1):
        print(f"\n{i}. Page {result['page_number']}, Chunk
{result['chunk_id']}")
        print(f"   Score: {result['score']:.4f}")

        if "highlights" in result:
            print("   Highlights:")
            for highlight in result["highlights"]:
                print(f"   ...{highlight}...")
        else:
            text = result["text"]
```

```
            snippet = text[:200] + "..." if len(text) > 200 else text
            print(f"   Text: {snippet}")

if __name__ == "__main__":
    asyncio.run(main())
```

## 9. Azure AI Search - Managed Document Processing

Azure AI Search now provides more built-in capabilities for document processing through its integration with Azure AI Services. Here's how you can use the Azure portal to set up a document chunking and vectorization pipeline:

1. **Create an AI Search service in Azure Portal**
2. **Set up Cognitive Search:**
    - Go to your search service
    - Select "Import data" wizard
    - Connect to your data source (Blob Storage, etc.)
    - Add cognitive skills:
        - OCR (for scanned documents)
        - Document cracking (for PDFs, Office docs)
        - Text splitting
        - Language detection
        - Key phrase extraction
        - Entity recognition
        - Vector embedding generation (requires Azure OpenAI connection)
3. **Configure chunking parameters:**
    - In the Text Split skill configuration:
        - Set maximum document segments
        - Set maximum characters per page
        - Choose split mode (pages, sentences, paragraphs)
4. **Add vectorization:**
    - Azure AI Search integrates with Azure OpenAI for vector embeddings
    - Configure the vector search parameters in your index
    - Set up the vector fields and dimensions
5. **Run the indexer** to process your documents

## 10. Summary: Best Practices for Document Chunking and Vectorization

1. **Document Preparation:**
    - Use high-quality PDFs or office documents
    - Ensure documents have proper text encoding (not scanned images without OCR)
    - Organize documents in logical containers in blob storage

2. **Chunking Strategies:**
    - Balance chunk size: 500-1000 tokens is often optimal
    - Consider semantic boundaries (paragraphs, sections) when splitting
    - Use overlapping chunks to maintain context across boundaries
    - Adjust chunking strategy based on document type (technical, narrative, etc.)
3. **Vectorization:**
    - Use Azure OpenAI's text-embedding models for best results
    - Consider model dimensions (1536 for ada-002)
    - Ensure your vector search configuration matches your embedding model
4. **Performance Optimization:**
    - Process documents in batches
    - Use Azure Functions for high-volume processing
    - Consider hybrid search (combination of keyword and vector)
    - Use filters to narrow search scope before vector search
5. **Error Handling:**
    - Implement robust error handling for document processing
    - Log processing errors for each document
    - Consider retry mechanisms for transient failures

By implementing these approaches, you can create a powerful document processing pipeline that automatically breaks down complete documents into searchable chunks with vector embeddings for semantic search capabilities.

Would you like me to expand on any specific part of this implementation or provide more details on a particular aspect of document chunking and vectorization with Azure AI Search?