I'll guide you through creating and using Azure Blob Storage resources programmatically for the document processing pipeline. This will integrate seamlessly with the document chunking and vectorization process we discussed earlier.

Creating and Managing Azure Blob Storage Resources with Python

1. Setting Up Azure Blob Storage Resources

First, let's create a comprehensive function to set up all the necessary storage resources:

```
from azure.identity import AzureCliCredential
from azure.mgmt.resource import ResourceManagementClient
from azure.mgmt.storage import StorageManagementClient
from azure.storage.blob import BlobServiceClient, BlobClient,
ContainerClient
import os
import time

def create_storage_resources(resource_group_name, location,
storage_account_name=None, container_name="documents"):
    """
    Create or get Azure Storage resources needed for document processing

    Returns:
        dict: Configuration details for the storage resources
    """
    # Set default storage account name if not provided
    if storage_account_name is None:
        # Generate a unique name using timestamp
        timestamp = int(time.time())
        storage_account_name = f"docstorage{timestamp}"
        # Ensure it's lowercase and alphanumeric
        storage_account_name = storage_account_name.lower()[:24]

    print(f"Setting up storage account: {storage_account_name}")

    # Authenticate using Azure CLI credentials
    credential = AzureCliCredential()

    # Create Resource Management client
    resource_client = ResourceManagementClient(credential,
os.environ.get("AZURE_SUBSCRIPTION_ID"))

    # Check if resource group exists, create if not
    try:
        resource_client.resource_groups.get(resource_group_name)
        print(f"Resource group '{resource_group_name}' already exists.")
    except Exception:
        print(f"Creating resource group '{resource_group_name}'...")
        resource_client.resource_groups.create_or_update(
            resource_group_name,
            {"location": location}
        )

    # Create Storage Management client
    storage_client = StorageManagementClient(credential,
os.environ.get("AZURE_SUBSCRIPTION_ID"))

    # Check if storage account exists
    storage_account_exists = False
    try:

storage_client.storage_accounts.get_properties(resource_group_name,
storage_account_name)
        storage_account_exists = True
        print(f"Storage account '{storage_account_name}' already exists.")
    except Exception:
        storage_account_exists = False

    # Create storage account if it doesn't exist
    if not storage_account_exists:
        print(f"Creating storage account '{storage_account_name}'...")
        poller = storage_client.storage_accounts.begin_create(
            resource_group_name,
            storage_account_name,
            {
                "location": location,
                "kind": "StorageV2",
                "sku": {"name": "Standard_LRS"}
            }
        )
        # Wait for the operation to complete
        account_result = poller.result()
        print(f"Storage account '{storage_account_name}' created.")

    # Get storage account keys
    keys = storage_client.storage_accounts.list_keys(resource_group_name,
storage_account_name)
    storage_key = keys.keys[0].value

    # Create connection string
    connection_string = f"DefaultEndpointsProtocol=https;AccountName=
{storage_account_name};AccountKey=
{storage_key};EndpointSuffix=core.windows.net"

    # Create Blob Service client
    blob_service_client =
BlobServiceClient.from_connection_string(connection_string)

    # Create container if it doesn't exist
    try:
        container_client =
blob_service_client.get_container_client(container_name)
        container_exists = container_client.exists()
        if not container_exists:
            print(f"Creating container '{container_name}'...")
            blob_service_client.create_container(container_name)
        else:
            print(f"Container '{container_name}' already exists.")
    except Exception as e:
        print(f"Creating container '{container_name}'...")
        blob_service_client.create_container(container_name)

    # Return the storage information
    storage_info = {
        "resource_group_name": resource_group_name,
        "storage_account_name": storage_account_name,
        "storage_account_key": storage_key,
        "connection_string": connection_string,
        "container_name": container_name,
        "blob_endpoint":
f"https://{storage_account_name}.blob.core.windows.net"
    }

    print(f"Storage resources configured successfully!")
    return storage_info
```

2. Uploading Documents to Blob Storage

Let's create functions to handle document uploads and management:

```
def upload_document(connection_string, container_name, local_file_path,
blob_name=None):
    """
    Upload a document to Azure Blob Storage

    Args:
        connection_string: Storage account connection string
        container_name: Container name
        local_file_path: Path to local file
        blob_name: Optional custom name for the blob

    Returns:
        str: URL of the uploaded blob
    """
    # Create BlobServiceClient
    blob_service_client =
BlobServiceClient.from_connection_string(connection_string)

    # Get or create a container
    container_client =
blob_service_client.get_container_client(container_name)

    # Use file name if blob_name not provided
    if blob_name is None:
        blob_name = os.path.basename(local_file_path)
    # Create blob client
    blob_client = container_client.get_blob_client(blob_name)

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

    print(f"Uploaded: {local_file_path} -> {container_name}/{blob_name}")

    # Return the blob URL
    account_name = blob_service_client.account_name
    return
f"https://{account_name}.blob.core.windows.net/{container_name}/{blob_name
}"

def upload_multiple_documents(connection_string, container_name,
directory_path, recursive=True):
    """
    Upload multiple documents from a directory to Azure Blob Storage

    Args:
        connection_string: Storage account connection string
        container_name: Container name
        directory_path: Path to local directory containing files
        recursive: Whether to search subdirectories

    Returns:
        list: URLs of all uploaded blobs
    """
    # Create BlobServiceClient
    blob_service_client =
BlobServiceClient.from_connection_string(connection_string)

    # Get or create a container
    container_client =
blob_service_client.get_container_client(container_name)

    # Find all files
    all_files = []
    if recursive:
        for root, dirs, files in os.walk(directory_path):
            for file in files:
                all_files.append(os.path.join(root, file))
    else:
        all_files = [os.path.join(directory_path, f) for f in
os.listdir(directory_path)
                     if os.path.isfile(os.path.join(directory_path, f))]

    # Upload each file
    uploaded_urls = []
    for file_path in all_files:
        # Create relative path for blob name
        rel_path = os.path.relpath(file_path, directory_path)
        rel_path = rel_path.replace("\\", "/")  # Ensure forward slashes
for blob names

        # Upload the file
        url = upload_document(connection_string, container_name,
file_path, rel_path)
        uploaded_urls.append(url)

    return uploaded_urls
```

3. Listing and Managing Blobs

Let's add functions to list, download, and delete blobs:

```
def list_blobs(connection_string, container_name, prefix=None):
    """
    List all blobs in a container, optionally filtered by prefix

    Args:
        connection_string: Storage account connection string
        container_name: Container name
        prefix: Optional prefix to filter blobs

    Returns:
        list: Information about blobs
    """
    # Create BlobServiceClient
    blob_service_client =
BlobServiceClient.from_connection_string(connection_string)

    # Get container client
    container_client =
blob_service_client.get_container_client(container_name)

    # List blobs
    blob_list = container_client.list_blobs(name_starts_with=prefix)

    # Convert to a more usable format
    blobs = []
    for blob in blob_list:
        blob_client = container_client.get_blob_client(blob.name)
        blob_url = blob_client.url

        # Get properties
        properties = blob_client.get_blob_properties()

        blobs.append({
            "name": blob.name,
            "url": blob_url,
            "size": properties.size,
            "content_type": properties.content_settings.content_type,
            "created_on": properties.creation_time,
            "last_modified": properties.last_modified
        })

    return blobs

def download_blob(connection_string, container_name, blob_name,
destination_file_path):
    """
    Download a blob to a local file

    Args:
        connection_string: Storage account connection string
        container_name: Container name
        blob_name: Name of the blob
        destination_file_path: Local file path to save the blob
    """
    # Create BlobServiceClient
    blob_service_client =
BlobServiceClient.from_connection_string(connection_string)

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

    # Download the blob
    with open(destination_file_path, "wb") as download_file:
        download_file.write(blob_client.download_blob().readall())

    print(f"Downloaded: {blob_name} -> {destination_file_path}")

def delete_blob(connection_string, container_name, blob_name):
    """
    Delete a blob

    Args:
        connection_string: Storage account connection string
        container_name: Container name
        blob_name: Name of the blob
    """
    # Create BlobServiceClient
    blob_service_client =
BlobServiceClient.from_connection_string(connection_string)

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

    # Delete the blob
    blob_client.delete_blob()

    print(f"Deleted blob: {blob_name}")
```

4. Setting Metadata and Properties

For more advanced scenarios, you might want to set metadata on your blobs:

```
def set_blob_metadata(connection_string, container_name, blob_name,
metadata):
    """
    Set metadata on a blob

    Args:
        connection_string: Storage account connection string
        container_name: Container name
        blob_name: Name of the blob
        metadata: Dictionary of metadata key-value pairs
    """
    # Create BlobServiceClient
    blob_service_client =
BlobServiceClient.from_connection_string(connection_string)

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

    # Set metadata
    blob_client.set_blob_metadata(metadata)

    print(f"Set metadata on blob: {blob_name}")

def set_blob_properties(connection_string, container_name, blob_name,
content_type=None, content_language=None):
    """
    Set properties on a blob

    Args:
        connection_string: Storage account connection string
        container_name: Container name
        blob_name: Name of the blob
        content_type: Content type of the blob
        content_language: Content language of the blob
    """
    # Create BlobServiceClient
    blob_service_client =
BlobServiceClient.from_connection_string(connection_string)

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

    # Get current properties
    properties = blob_client.get_blob_properties().content_settings

    # Update content settings
    blob_client.set_http_headers(
        content_settings={
            "content_type": content_type or properties.content_type,
            "content_language": content_language or
properties.content_language,
            "cache_control": properties.cache_control,
            "content_encoding": properties.content_encoding,
            "content_disposition": properties.content_disposition,
            "content_md5": properties.content_md5
        }
    )

    print(f"Set properties on blob: {blob_name}")
```

5. Generating SAS Tokens for Secure Access

Sometimes you'll need to generate SAS (Shared Access Signature) tokens for temporary access:

```python
from datetime import datetime, timedelta

def generate_blob_sas_token(
        connection_string,
        container_name,
        blob_name,
        permission="r",
        expiry_hours=1):
    """
    Generate a SAS token for a blob

    Args:
        connection_string: Storage account connection string
        container_name: Container name
        blob_name: Name of the blob
        permission: Permissions (r=read, w=write, d=delete, l=list)
        expiry_hours: Token expiry in hours

    Returns:
        str: SAS token URL for the blob
    """
    from azure.storage.blob import generate_blob_sas, BlobSasPermissions

    # Parse the connection string
    from azure.storage.blob._shared.shared_access_signature import
BlobSharedAccessSignature
    from azure.storage.blob._shared.parse_connection_string import
parse_connection_string
    conn_dict = parse_connection_string(connection_string)
    account_name = conn_dict['AccountName']
    account_key = conn_dict['AccountKey']

    # Set permission
    permission = BlobSasPermissions(
        read='r' in permission,
        write='w' in permission,
        delete='d' in permission,
        list='l' in permission
    )

    # Generate SAS token
    sas_token = generate_blob_sas(
        account_name=account_name,
        container_name=container_name,
        blob_name=blob_name,
        account_key=account_key,
        permission=permission,
        expiry=datetime.utcnow() + timedelta(hours=expiry_hours)
    )

    # Return full URL
    return
f"https://{account_name}.blob.core.windows.net/{container_name}/{blob_name
}?{sas_token}"
```

6. Complete Example: Storage Setup and Document Processing Integration

Now, let's put everything together to create a complete workflow for document processing with Azure Blob Storage:

```
def main():
    """
    Main function to demonstrate the complete document processing workflow
with storage
    """
    import os

    # Set up Azure environment variables if not already set
    if "AZURE_SUBSCRIPTION_ID" not in os.environ:
        # This would normally come from environment, authentication, or
user input
        os.environ["AZURE_SUBSCRIPTION_ID"] = input("Enter your Azure
Subscription ID: ")

    # Get configuration parameters
    resource_group_name = input("Enter your resource group name: ")
    location = input("Enter location (e.g., eastus): ")

    # 1. Set up storage resources
    print("\n=== Setting up Azure Storage Resources ===")
    storage_info = create_storage_resources(
        resource_group_name=resource_group_name,
        location=location
    )

    # Display storage information
    print("\nStorage Account Information:")
    print(f"Account Name: {storage_info['storage_account_name']}")
    print(f"Container: {storage_info['container_name']}")
    print(f"Blob Endpoint: {storage_info['blob_endpoint']}")

    # 2. Upload documents
    print("\n=== Document Upload Options ===")
    print("1. Upload a single document")
    print("2. Upload all documents from a directory")

    choice = input("Enter your choice (1 or 2): ")

    if choice == "1":
        local_file_path = input("Enter local file path: ")
        blob_url = upload_document(
            connection_string=storage_info['connection_string'],
            container_name=storage_info['container_name'],
            local_file_path=local_file_path
        )
        print(f"Document uploaded: {blob_url}")

    elif choice == "2":
        directory_path = input("Enter directory path: ")
        recursive = input("Include subdirectories? (y/n): ").lower() ==
'y'

        blob_urls = upload_multiple_documents(
            connection_string=storage_info['connection_string'],
            container_name=storage_info['container_name'],
            directory_path=directory_path,
            recursive=recursive
        )

        print(f"\nUploaded {len(blob_urls)} documents")

    # 3. List uploaded documents
    print("\n=== Listing Uploaded Documents ===")
    blobs = list_blobs(
        connection_string=storage_info['connection_string'],
        container_name=storage_info['container_name']
    )

    for i, blob in enumerate(blobs, 1):
        print(f"{i}. {blob['name']} ({blob['content_type']},
{blob['size']} bytes)")

    # 4. Now process documents with the AI Search pipeline
    print("\n=== Setting up Document Processing Pipeline ===")
    # Get additional required parameters
    search_service_name = input("Enter your AI Search service name: ")
    search_admin_key = input("Enter your AI Search admin key: ")

    # Choose the embedding service (OpenAI or Azure OpenAI)
    print("\nEmbedding Service Options:")
    print("1. OpenAI")
    print("2. Azure OpenAI")

    embedding_choice = input("Choose embedding service (1 or 2): ")

    # Get embedding service parameters
    if embedding_choice == "1":
        openai_api_key = input("Enter your OpenAI API key: ")
        form_recognizer_endpoint = input("Enter your Form Recognizer 
endpoint: ")
        form_recognizer_key = input("Enter your Form Recognizer key: ")

        # Set up pipeline with standard OpenAI
        from previous_code import setup_document_processing_pipeline, 
create_chunked_document_index

        pipeline_config = setup_document_processing_pipeline(
            resource_group_name=resource_group_name,
            storage_account_name=storage_info['storage_account_name'],
            storage_account_key=storage_info['storage_account_key'],
            search_service_name=search_service_name,
            search_admin_key=search_admin_key,
            form_recognizer_endpoint=form_recognizer_endpoint,
            form_recognizer_key=form_recognizer_key,
            openai_api_key=openai_api_key
        )

    else:  # Azure OpenAI
        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: ")
        form_recognizer_endpoint = input("Enter your Form Recognizer 
endpoint: ")
        form_recognizer_key = input("Enter your Form Recognizer key: ")

        # Set up pipeline with Azure OpenAI
        from previous_code import setup_document_processing_pipeline, 
create_chunked_document_index

        pipeline_config = setup_document_processing_pipeline(
            resource_group_name=resource_group_name,
            storage_account_name=storage_info['storage_account_name'],
            storage_account_key=storage_info['storage_account_key'],
            search_service_name=search_service_name,
            search_admin_key=search_admin_key,
            form_recognizer_endpoint=form_recognizer_endpoint,
            form_recognizer_key=form_recognizer_key,
            azure_openai_endpoint=azure_openai_endpoint,
            azure_openai_key=azure_openai_key,
            azure_openai_deployment=azure_openai_deployment
        )

    # Create the search index
    index_name = "chunked-documents"
    create_chunked_document_index(pipeline_config["index_client"],
index_name)

    # 5. Process each document
    print("\n=== Processing Documents ===")

    # Choose processing options
    print("Document Processing Options:")
    print("1. Process a single document")
    print("2. Process all uploaded documents")

    process_choice = input("Choose option (1 or 2): ")

    from previous_code import process_and_index_document

    if process_choice == "1":
        # Let user select a document from the uploaded list
        for i, blob in enumerate(blobs, 1):
            print(f"{i}. {blob['name']}")

        blob_index = int(input("Enter document number to process: ")) - 1
        selected_blob = blobs[blob_index]

        # Download the blob to a temporary file for processing
        import tempfile
        with tempfile.NamedTemporaryFile(delete=False,
suffix=os.path.splitext(selected_blob['name'])[1]) as temp_file:
            temp_path = temp_file.name

        download_blob(
            connection_string=storage_info['connection_string'],
            container_name=storage_info['container_name'],
            blob_name=selected_blob['name'],
            destination_file_path=temp_path
        )

        # Process the document
        document_id = process_and_index_document(
            temp_path,
            pipeline_config,
            index_name=index_name
        )

        # Clean up the temporary file
        os.unlink(temp_path)
```

```
            print(f"Document processed with ID: {document_id}")

        else:  # Process all documents
            # Process each document
            document_ids = []

            for blob in blobs:
                print(f"Processing {blob['name']}...")

                # Download the blob to a temporary file
                import tempfile
                with tempfile.NamedTemporaryFile(delete=False,
suffix=os.path.splitext(blob['name'])[1]) as temp_file:
                    temp_path = temp_file.name

                download_blob(
                    connection_string=storage_info['connection_string'],
                    container_name=storage_info['container_name'],
                    blob_name=blob['name'],
                    destination_file_path=temp_path
                )

                # Process the document
                document_id = process_and_index_document(
                    temp_path,
                    pipeline_config,
                    index_name=index_name
                )

                document_ids.append(document_id)

                # Clean up the temporary file
                os.unlink(temp_path)

            print(f"Processed {len(document_ids)} documents")

        # 6. Search the documents
        from previous_code import semantic_search_document_chunks

        # Create search client
        from azure.search.documents import SearchClient
        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"]
        )

        # 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()
```

7. Handling Large Document Uploads with Block Blobs

For large documents, you'll want to use block blob uploads with progress tracking:

```
def upload_large_document(connection_string, container_name, 
local_file_path, blob_name=None):
    """
    Upload a large document to Azure Blob Storage using block blobs with 
progress tracking

    Args:
        connection_string: Storage account connection string
        container_name: Container name
        local_file_path: Path to local file
        blob_name: Optional custom name for the blob

    Returns:
        str: URL of the uploaded blob
    """
    # Create BlobServiceClient
    blob_service_client =
BlobServiceClient.from_connection_string(connection_string)

    # Use file name if blob_name not provided
    if blob_name is None:
        blob_name = os.path.basename(local_file_path)

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

    # Get file size
    file_size = os.path.getsize(local_file_path)

    # Define a callback to track progress
    def progress_callback(current, total):
        progress = (current / total) * 100
        print(f"Upload progress: {progress:.2f}%", end="\r")

    # Set the chunk size (4MB is a good default)
    chunk_size = 4 * 1024 * 1024  # 4MB

    # Upload the file using blocks
    with open(local_file_path, "rb") as data:
        # Calculate number of chunks
        num_chunks = (file_size + chunk_size - 1) // chunk_size

        # Create a list to hold block IDs
        block_list = []

        for i in range(num_chunks):
            # Read a chunk of data
            chunk = data.read(chunk_size)
            if not chunk:
                break

            # Generate block ID (must be base64 encoded)
            import base64
            block_id =
base64.b64encode(str(i).zfill(10).encode()).decode()

            # Upload the block
            blob_client.stage_block(
                block_id=block_id,
                data=chunk,
                length=len(chunk)
            )

            # Add the block ID to the list
            block_list.append(block_id)
            # Update progress
            progress_callback(min((i + 1) * chunk_size, file_size),
file_size)

        # Commit the blocks
        blob_client.commit_block_list(block_list)

    print(f"Uploaded: {local_file_path} -> {container_name}/{blob_name}")

    # Return the blob URL
    account_name = blob_service_client.account_name
    return
f"https://{account_name}.blob.core.windows.net/{container_name}/{blob_name
}"
```

8. Complete Code Example for Document Processing

Here's a concise example that shows how to set up storage and process a document in one sequence:

```
def process_document_from_file(
        resource_group_name,
        location,
        search_service_name,
        search_admin_key,
        form_recognizer_endpoint,
        form_recognizer_key,
        openai_api_key,
        local_file_path):
    """
    Complete end-to-end process to upload, chunk, vectorize, and index a
document
    """
    # 1. Create storage resources
    storage_info = create_storage_resources(
        resource_group_name=resource_group_name,
        location=location
    )

    # 2. Upload document
    blob_url = upload_document(
        connection_string=storage_info['connection_string'],
        container_name=storage_info['container_name'],
        local_file_path=local_file_path
    )

    # 3. Setup processing pipeline
    from previous_code import setup_document_processing_pipeline,
create_chunked_document_index

    pipeline_config = setup_document_processing_pipeline(
        resource_group_name=resource_group_name,
        storage_account_name=storage_info['storage_account_name'],
        storage_account_key=storage_info['storage_account_key'],
        search_service_name=search_service_name,
        search_admin_key=search_admin_key,
        form_recognizer_endpoint=form_recognizer_endpoint,
        form_recognizer_key=form_recognizer_key,
        openai_api_key=openai_api_key
    )

    # 4. Create search index
    index_name = "chunked-documents"
    create_chunked_document_index(pipeline_config["index_client"],
index_name)

    # 5. Process and index document
    from previous_code import process_and_index_document

    document_id = process_and_index_document(
        local_file_path,
        pipeline_config,
        index_name=index_name
    )

    # 6. Return details
    return {
        "document_id": document_id,
        "blob_url": blob_url,
        "storage_info": storage_info,
        "index_name": index_name
    }
```

Integration with Azure AI Search

Finally, here's a function that ties together the storage components with the AI Search pipeline:

```
def setup_complete_document_processing_system(
        resource_group_name,
        location,
        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):
    """
    Set up a complete document processing system with storage and search

    Returns:
        dict: Configuration for the complete system
    """
    # 1. Create storage resources
    storage_info = create_storage_resources(
        resource_group_name=resource_group_name,
        location=location
    )

    # 2. Setup processing pipeline
    from previous_code import setup_document_processing_pipeline,
create_chunked_document_index

    pipeline_config = setup_document_processing_pipeline(
        resource_group_name=resource_group_name,
        storage_account_name=storage_info['storage_account_name'],
        storage_account_key=storage_info['storage_account_key'],
        search_service_name=search_service_name,
        search_admin_key=search_admin_key,
        form_recognizer_endpoint=form_recognizer_endpoint,
        form_recognizer_key=form_recognizer_key,
        openai_api_key=openai_api_key,
        azure_openai_endpoint=azure_openai_endpoint,
        azure_openai_key=azure_openai_key,
        azure_openai_deployment=azure_openai_deployment
    )

    # 3. Create search index
    index_name = "chunked-documents"
    create_chunked_document_index(pipeline_config["index_client"],
index_name)

    # 4. Return complete system configuration
    system_config = {
        "storage": storage_info,
        "pipeline": pipeline_config,
        "index_name": index_name,
        "upload_function": upload_document,
        "process_function": process_and_index_document,
        "search_function": semantic_search_document_chunks
    }

    return system_config
```

9. Document Processing API

Let's create a simple API that allows for programmatic document processing:

```
class DocumentProcessingAPI:
    """
    A class providing an API for document processing operations
    """
    def __init__(self, system_config):
        """Initialize with system configuration"""
        self.system_config = system_config
        self.storage_info = system_config["storage"]
        self.pipeline_config = system_config["pipeline"]
        self.index_name = system_config["index_name"]

    def upload_document(self, local_file_path, blob_name=None):
        """Upload a document to storage"""
        return upload_document(
            connection_string=self.storage_info["connection_string"],
            container_name=self.storage_info["container_name"],
            local_file_path=local_file_path,
            blob_name=blob_name
        )

    def process_document(self, local_file_path, tags=None):
        """Process and index a document"""
        from previous_code import process_and_index_document

        return process_and_index_document(
            local_file_path,
            self.pipeline_config,
            index_name=self.index_name,
            tags=tags
        )

    def upload_and_process_document(self, local_file_path, blob_name=None,
tags=None):
        """Upload and process a document in one step"""
        # Upload
        blob_url = self.upload_document(local_file_path, blob_name)

        # Process
        document_id = self.process_document(local_file_path, tags)

        return {
            "document_id": document_id,
            "blob_url": blob_url
        }

    def search_documents(self, query_text, document_id=None, top=10):
        """Search for document chunks"""
        # Create search client
        from azure.search.documents import SearchClient
        search_client = SearchClient(
            endpoint=self.pipeline_config["search_endpoint"],
            index_name=self.index_name,
            credential=self.pipeline_config["search_credential"]
        )

        # Perform search
        from previous_code import semantic_search_document_chunks

        results = semantic_search_document_chunks(
            search_client,
            query_text,
            self.pipeline_config["embedding_source"],
            self.pipeline_config["azure_openai_deployment"],
            document_id=document_id,
            top=top
        )

        return results

    def list_documents(self):
        """List all documents in storage"""
        return list_blobs(
            connection_string=self.storage_info["connection_string"],
            container_name=self.storage_info["container_name"]
        )

    def delete_document(self, blob_name):
        """Delete a document from storage"""
        return delete_blob(
            connection_string=self.storage_info["connection_string"],
            container_name=self.storage_info["container_name"],
            blob_name=blob_name
        )
```

10. Creating a Complete Command-Line Interface

Finally, let's create a complete command-line interface for this document processing system:

```
def create_cli():
    """Create a command-line interface for document processing"""
    import argparse

    parser = argparse.ArgumentParser(description="Document Processing
CLI")
    subparsers = parser.add_subparsers(dest="command", help="Command to
run")

    # Setup command
    setup_parser = subparsers.add_parser("setup", help="Set up the
document processing system")
    setup_parser.add_argument("--resource-group", required=True,
help="Azure resource group name")
    setup_parser.add_argument("--location", required=True, help="Azure
location (e.g., eastus)")
    setup_parser.add_argument("--search-service", required=True,
help="Azure AI Search service name")
    setup_parser.add_argument("--search-key", required=True, help="Azure
AI Search admin key")
    setup_parser.add_argument("--form-recognizer-endpoint", required=True,
help="Form Recognizer endpoint")
```

```
    setup_parser.add_argument("--form-recognizer-key", required=True,
help="Form Recognizer key")
    setup_parser.add_argument("--openai-key", help="OpenAI API key
(optional)")
    setup_parser.add_argument("--azure-openai-endpoint", help="Azure
OpenAI endpoint (optional)")
    setup_parser.add_argument("--azure-openai-key", help="Azure OpenAI key
(optional)")
    setup_parser.add_argument("--azure-openai-deployment", help="Azure
OpenAI deployment name (optional)")

    # Upload command
    upload_parser = subparsers.add_parser("upload", help="Upload a
document")
    upload_parser.add_argument("--config", required=True, help="Path to
saved configuration file")
    upload_parser.add_argument("--file", required=True, help="Path to
document file")
    upload_parser.add_argument("--name", help="Custom blob name
(optional)")

    # Process command
    process_parser = subparsers.add_parser("process", help="Process and
index a document")
    process_parser.add_argument("--config", required=True, help="Path to
saved configuration file")
    process_parser.add_argument("--file", required=True, help="Path to
document file")
    process_parser.add_argument("--tags", help="Comma-separated tags
(optional)")

    # Upload-and-process command
    upload_process_parser = subparsers.add_parser("upload-process",
help="Upload and process a document")
    upload_process_parser.add_argument("--config", required=True,
help="Path to saved configuration file")
    upload_process_parser.add_argument("--file", required=True, help="Path
to document file")
    upload_process_parser.add_argument("--name", help="Custom blob name
(optional)")
    upload_process_parser.add_argument("--tags", help="Comma-separated
tags (optional)")

    # Search command
    search_parser = subparsers.add_parser("search", help="Search
documents")
    search_parser.add_argument("--config", required=True, help="Path to
saved configuration file")
    search_parser.add_argument("--query", required=True, help="Search
query")
    search_parser.add_argument("--document-id", help="Limit search to
specific document ID (optional)")
    search_parser.add_argument("--top", type=int, default=10, help="Number
of results to return")
```

```
    # List command
    list_parser = subparsers.add_parser("list", help="List all documents")
    list_parser.add_argument("--config", required=True, help="Path to saved configuration file")

    # Delete command
    delete_parser = subparsers.add_parser("delete", help="Delete a document")
    delete_parser.add_argument("--config", required=True, help="Path to saved configuration file")
    delete_parser.add_argument("--name", required=True, help="Blob name to delete")

    return parser

def run_cli():
    """Run the command-line interface"""
    import json
    import os

    # Create parser
    parser = create_cli()
    args = parser.parse_args()

    if args.command == "setup":
        # Set up the system
        system_config = setup_complete_document_processing_system(
            resource_group_name=args.resource_group,
            location=args.location,
            search_service_name=args.search_service,
            search_admin_key=args.search_key,
            form_recognizer_endpoint=args.form_recognizer_endpoint,
            form_recognizer_key=args.form_recognizer_key,
            openai_api_key=args.openai_key,
            azure_openai_endpoint=args.azure_openai_endpoint,
            azure_openai_key=args.azure_openai_key,
            azure_openai_deployment=args.azure_openai_deployment
        )

        # Save the configuration
        config_file = "document_processing_config.json"

        # We cannot serialize the function objects, so remove them
        serializable_config = {
            "storage": system_config["storage"],
            "pipeline": {
                k: v for k, v in system_config["pipeline"].items()
                if not callable(v) and not str(k).startswith("_")
            },
            "index_name": system_config["index_name"]
        }

        with open(config_file, "w") as f:
            json.dump(serializable_config, f, indent=2)

        print(f"System set up successfully. Configuration saved to
{config_file}")

    else:
        # Load configuration
        if not os.path.exists(args.config):
            print(f"Error: Configuration file {args.config} not found")
            return

        with open(args.config, "r") as f:
            config = json.load(f)

        # Create API object
        api = DocumentProcessingAPI(config)

        if args.command == "upload":
            # Upload a document
            blob_url = api.upload_document(args.file, args.name)
            print(f"Document uploaded: {blob_url}")

        elif args.command == "process":
            # Process a document
            tags = args.tags.split(",") if args.tags else None
            document_id = api.process_document(args.file, tags)
            print(f"Document processed with ID: {document_id}")

        elif args.command == "upload-process":
            # Upload and process a document
            tags = args.tags.split(",") if args.tags else None
            result = api.upload_and_process_document(args.file, args.name,
tags)
            print(f"Document uploaded and processed:")
            print(f"  Document ID: {result['document_id']}")
            print(f"  Blob URL: {result['blob_url']}")

        elif args.command == "search":
            # Search documents
            results = api.search_documents(args.query, args.document_id,
args.top)

            print(f"Found {len(results)} results for query:
'{args.query}'")
            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 or snippet
                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}")

        elif args.command == "list":
            # List documents
            blobs = api.list_documents()

            print(f"Found {len(blobs)} documents:")
            for i, blob in enumerate(blobs, 1):
                print(f"{i}. {blob['name']} ({blob['content_type']},
{blob['size']} bytes)")
                print(f"   Last modified: {blob['last_modified']}")
                print(f"   URL: {blob['url']}")

        elif args.command == "delete":
            # Delete a document
            api.delete_document(args.name)
            print(f"Document {args.name} deleted")

        else:
            parser.print_help()

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

Putting It All Together: A Complete Example

Here's a complete example of how to use this system to process a document from scratch, including setting up storage resources:

```
def complete_demo():
    """Complete demo showing the entire document processing workflow"""
    import os

    # Configuration
    resource_group_name = "doc-processing-rg"
    location = "eastus"
    search_service_name = "your-search-service-name"
    search_admin_key = "your-search-admin-key"
    form_recognizer_endpoint = "https://your-form-
recognizer.cognitiveservices.azure.com/"
    form_recognizer_key = "your-form-recognizer-key"
    openai_api_key = "your-openai-key"
    local_document_path = "/path/to/your/document.pdf"

    # Set subscription ID
    os.environ["AZURE_SUBSCRIPTION_ID"] = "your-subscription-id"
    # 1. Set up storage resources
    print("Setting up storage resources...")
    storage_info = create_storage_resources(
        resource_group_name=resource_group_name,
        location=location
    )

    # 2. Upload document
    print("Uploading document...")
    blob_url = upload_document(
        connection_string=storage_info['connection_string'],
        container_name=storage_info['container_name'],
        local_file_path=local_document_path
    )

    # 3. Set up document processing pipeline
    print("Setting up document processing pipeline...")
    pipeline_config = setup_document_processing_pipeline(
        resource_group_name=resource_group_name,
        storage_account_name=storage_info['storage_account_name'],
        storage_account_key=storage_info['storage_account_key'],
        search_service_name=search_service_name,
        search_admin_key=search_admin_key,
        form_recognizer_endpoint=form_recognizer_endpoint,
        form_recognizer_key=form_recognizer_key,
        openai_api_key=openai_api_key
    )

    # 4. Create search index
    print("Creating search index...")
    index_name = "chunked-documents"
    create_chunked_document_index(pipeline_config["index_client"],
index_name)

    # 5. Process document
    print("Processing document...")
    document_id = process_and_index_document(
        local_document_path,
        pipeline_config,
        index_name=index_name,
        tags=["demo", "sample"]
    )

    # 6. Search for content
    print("\nSearching for content...")

    # Create search client
    from azure.search.documents import SearchClient
    search_client = SearchClient(
        endpoint=pipeline_config["search_endpoint"],
        index_name=index_name,
        credential=pipeline_config["search_credential"]
    )
    # Perform searches
    sample_queries = [
        "What is the main topic of this document?",
        "Can you summarize the key points?",
        "What are the recommendations in this document?"
    ]

    for query in sample_queries:
        print(f"\nQuery: '{query}'")

        results = semantic_search_document_chunks(
            search_client,
            query,
            pipeline_config["embedding_source"],
            pipeline_config["azure_openai_deployment"],
            document_id=document_id,
            top=3
        )

        print(f"Found {len(results)} results:")
        for i, result in enumerate(results, 1):
            print(f"\n{i}. From page {result['page_number']}")
            print(f"   Score: {result['score']:.4f}")

            # Show highlights or snippet
            if "highlights" in result:
                print("   Highlights:")
                for highlight in result["highlights"]:
                    print(f"   ...{highlight}...")
            else:
                text = result["text"]
                snippet = text[:150] + "..." if len(text) > 150 else text
                print(f"   Text: {snippet}")

    print("\nComplete document processing demo finished successfully!")
```

This comprehensive guide should provide you with all the tools you need to programmatically set up and manage Azure Blob Storage resources for your document processing pipeline, including chunking and vectorization. You can adapt and extend these examples based on your specific requirements.