S3 Document Reader/Writer Implementation Plan

This implementation plan provides a step-by-step guide for building a comprehensive tool that can read from and write to various document types stored in AWS S3 buckets.

1. Setup Development Environment

Prerequisites

1. **Python Environment**
   - Install Python 3.9 or later
   - Set up a virtual environment:

     ```
     bash python -m venv s3doc-envsource s3doc-env/bin/activate # Linux/macOS# ors3doc-env\Scripts\activate # Windows
     ```

2. **Required Python Libraries**
   - Install core AWS library:

     ```
     bash pip install boto3
     ```

   - Install document processing libraries:

     ```
     bash pip install pandas PyPDF2 python-docx openpyxl
     ```

   - Install API framework and utilities:

     ```
     bash pip install flask flask-restful marshmallow pyjwt
     ```

2. AWS Configuration

1. **AWS Account Setup**
   - If you don't have an AWS account, create one at AWS Console
   - **Create IAM User for S3 Access**
   - Go to the AWS Management Console
   - Navigate to IAM service
   - Create a new user with programmatic access
   - Attach the "AmazonS3FullAccess" policy (or create a custom policy with minimal permissions)
   - Save the access key ID and secret access key
   - **Configure AWS Credentials**
   - Install AWS CLI: `pip install awscli`
   - Configure credentials:

     ```
     bash aws configure
     ```

   - Enter your access key ID and secret access key when prompted
   - Set default region (e.g., `us-east-1`)
   - **Create an S3 Bucket** (Optional - can also be created by the tool)
   - Using the AWS CLI:

     ```
     bash aws s3 mb s3://your-document-bucket
     ```

   - Or via the AWS Console: S3 service > Create bucket
   - Optionally enable versioning for the bucket

3. Create Project Structure

Create the following project structure:

```
s3-document-manager/ â”œâ”€â”€ app/ â”‚ â”œâ”€â”€ __init__.py # Initialize Flask app â”‚ â”œâ”€â”€ api/ # API routes â”‚ â”‚ â”œâ”€â”€ __init__.py â”‚ â”‚ â”œâ”€â”€ auth.py # Authentication routes â”‚ â”‚ â”œâ”€â”€ buckets.py # Bucket management â”‚ â”‚ â”œâ”€â”€ documents.py # Document operations â”‚ â”œâ”€â”€ services/ # Business logic â”‚ â”‚ â”œâ”€â”€ __init__.py â”‚ â”‚ â”œâ”€â”€ auth_service.py # Authentication service â”‚ â”‚ â”œâ”€â”€ bucket_service.py # Bucket management service â”‚ â”‚ â”œâ”€â”€ document_service.py # Document service â”‚ â”œâ”€â”€ utils/ # Utility functions â”‚ â”‚ â”œâ”€â”€ __init__.py â”‚ â”‚ â”œâ”€â”€ document_handlers/ # Document type handlers â”‚ â”‚ â”‚ â”œâ”€â”€ __init__.py â”‚ â”‚ â”‚ â”œâ”€â”€ pdf_handler.py â”‚ â”‚ â”‚ â”œâ”€â”€ docx_handler.py â”‚ â”‚ â”‚ â”œâ”€â”€ xlsx_handler.py â”‚ â”‚ â”‚ â”œâ”€â”€ csv_handler.py â”‚ â”‚ â”‚ â”œâ”€â”€ text_handler.py â”‚ â”‚ â”‚ â”œâ”€â”€ json_handler.py â”‚ â”‚ â”œâ”€â”€ helpers.py # Helper functions â”‚ â”œâ”€â”€ models/ # Data models â”‚ â”‚ â”œâ”€â”€ __init__.py â”‚ â”‚ â”œâ”€â”€ schemas.py # Marshmallow schemas â”‚ â”œâ”€â”€ config.py # Configuration â”œâ”€â”€ run.py # Application entry point â”œâ”€â”€ requirements.txt # Dependencies â”œâ”€â”€ tests/ # Test files â”‚ â”œâ”€â”€ __init__.py â”‚ â”œâ”€â”€ test_api.py â”œâ”€â”€ README.md # Documentation
```

4. Core Application Modules

Module 1: S3 Connection and Bucket Management

1. **Implementation Tasks**
   - Establish connection to AWS S3
   - Create functions for bucket operations:
     - Create/delete buckets
     - List buckets
     - Check if bucket exists
     - Enable/disable versioning

Module 2: Document Processing Utilities

1. **Implementation Tasks**
   - Create utility functions for different document types:
     - PDF (using PyPDF2)
     - Word documents (using python-docx)
     - Excel spreadsheets (using openpyxl)
     - CSV files (using pandas)
     - Text files
     - JSON files

Module 3: Document Reading

1. **Implementation Tasks**
   - Implement functions to:
     - Download documents from S3
     - Detect document type
     - Extract content based on document type
     - Return content in appropriate format

Module 4: Document Writing

1. **Implementation Tasks**
   - Implement functions to:
     - Create new documents
     - Update existing documents
     - Handle different document formats
     - Upload to S3

Module 5: Document Management

1. **Implementation Tasks**
   - Implement functions to:
     - List documents in a bucket
     - Search for documents
     - Delete documents
     - Copy/move documents
     - Handle document versions

Module 6: API Application

1. **Implementation Tasks**
    - Create RESTful API endpoints
    - Implement request validation
    - Set up authentication and authorization
    - Create response formatting middleware

5. API Service Implementation

Authentication and Security

1. **API Authentication**

    - Implement JWT-based authentication
    - Create a secure token generation and validation system
    - Set up user management (create, update, delete users)
    - Store API keys securely

    - **Authorization**

    - Implement role-based access control

    - Create middleware for checking permissions
    - Map users to buckets and documents

    - **Security Headers**

    - Set up CORS policies

    - Implement secure headers
    - Add request validation

API Routes and Endpoints

1. **Authentication Routes**

    - `/api/auth/login` - Authenticate user and return JWT
    - `/api/auth/refresh` - Refresh JWT token
    - `/api/auth/register` - Register new API user (admin only)
    - `/api/auth/users` - Manage users (admin only)

    - **Bucket Management Routes**

    - `/api/buckets` - List all accessible buckets (GET) or create new bucket (POST)

    - `/api/buckets/{bucket_name}` - Get bucket details (GET), update bucket (PUT), or delete bucket (DELETE)
    - `/api/buckets/{bucket_name}/versioning` - Enable/disable versioning

    - **Document Management Routes**

    - `/api/buckets/{bucket_name}/documents` - List documents (GET) or upload new document (POST)

    - `/api/buckets/{bucket_name}/documents/search` - Search documents by metadata or content
    - `/api/buckets/{bucket_name}/documents/{key}` - Get document content (GET), update document (PUT), or delete document (DELETE)
    - `/api/buckets/{bucket_name}/documents/{key}/versions` - List all versions of a document
    - `/api/buckets/{bucket_name}/documents/{key}/versions/{version_id}` - Get specific version of a document

    - **Document Operations Routes**

    - `/api/operations/copy` - Copy document between buckets/paths

    - `/api/operations/move` - Move document between buckets/paths
    - `/api/operations/convert` - Convert document from one format to another
    - `/api/operations/compare` - Compare two document versions

Implementation Examples

Example of a route handler for listing documents:

```python @document_bp.route('/buckets//documents', methods=['GET']) @jwt_required def list_documents(): prefix = request.args.get('prefix', '') max_keys = int(request.args.get('max_keys', 1000))

# Call document service to list documents
documents = document_service.list_documents(bucket_name, prefix, max_keys)

return jsonify({
    'success': True,
    'count': len(documents),
    'documents': documents
})

```

Example of a document content retrieval endpoint:

```python @document_bp.route('/buckets//documents/', methods=['GET']) @jwt_required def get_document(): version_id = request.args.get('version_id') format_output = request.args.get('format', 'raw')

# Call document service to get document content
content = document_service.read_document(bucket_name, key, version_id, format_output)

if format_output == 'download':
    # Return file for download
    return send_file(
        io.BytesIO(content),
        mimetype='application/octet-stream',
        as_attachment=True,
        attachment_filename=os.path.basename(key)
    )

return jsonify({
    'success': True,
    'bucket': bucket_name,
    'key': key,
    'version_id': version_id,
    'content': content
})

```

Request Validation and Response Formatting

1. **Request Validation**
    - Implement Marshmallow schemas for validating incoming requests
    - Create custom validators for different document types
    - Handle validation errors gracefully
    - **Response Formatting**
    - Create standardized response format
    - Implement pagination for list responses
    - Format errors consistently

Error Handling

1. **Global Error Handler**
    - Implement application-wide exception handling
    - Map exceptions to appropriate HTTP status codes
    - Provide detailed error messages for debugging
    - **S3-Specific Error Handling**
    - Handle AWS S3 errors gracefully
    - Provide user-friendly error messages
    - Log detailed information for troubleshooting

API Documentation

1. **Swagger/OpenAPI Documentation**
    - Create comprehensive API documentation
    - Include request/response examples
    - Document all error codes and messages
    - **Interactive API Explorer**
    - Implement Swagger UI for testing API endpoints
    - Allow trying out endpoints directly from documentation

6. Data Processing Features

Feature 1: Document Content Extraction

1. **Text Extraction**
    - Extract text content from various document formats
    - Handle formatting preservation when possible
    - Support multiple languages
    - **Structured Data Extraction**
    - Extract tabular data from spreadsheets and PDFs
    - Parse tables to JSON format
    - Handle header detection and column typing

Feature 2: Document Content Modification

1. **Text Replacement**
    - Modify document content without changing format
    - Support find-and-replace operations
    - Handle formatting preservation
    - **Document Merging**
    - Combine multiple documents into one
    - Support similar formats (e.g., multiple PDFs)
    - Preserve structure and formatting

Feature 3: Document Analysis

1. **Text Analytics**
    - Provide basic text statistics (word count, readability)
    - Extract entities and keywords
    - Generate text summaries
    - **Data Analysis**
    - Generate statistics for tabular data
    - Create data visualizations
    - Filter and query structured data

7. Advanced API Features

Feature 1: Batch Operations

1. **Implementation Tasks**
    - Create endpoints for batch processing:
        - `/api/batch/read` - Read multiple documents
        - `/api/batch/write` - Write multiple documents
        - `/api/batch/delete` - Delete multiple documents

- Handle partial success scenarios
- Provide detailed results for each item in batch

Feature 2: Background Processing

1. **Implementation Tasks**
   - Implement job queue for long-running operations
   - Create endpoints for job management:
     - `/api/jobs` - List all jobs
     - `/api/jobs/{job_id}` - Get job status
     - `/api/jobs/{job_id}/cancel` - Cancel job
   - Provide progress tracking and notifications

Feature 3: Content Search

1. **Implementation Tasks**
   - Implement full-text search within documents
   - Create dedicated search endpoint:
     - `/api/search` - Search across buckets and documents
   - Support advanced query operators
   - Implement result ranking and highlighting

Feature 4: Webhooks

1. **Implementation Tasks**
   - Create webhook registration and management:
     - `/api/webhooks` - Manage webhook subscriptions
   - Trigger notifications on document events
   - Support customizable payload formats

8. API Client Examples

Using cURL

```bash

# List all buckets

curl -X GET "http://localhost:5000/api/buckets" \ -H "Authorization: Bearer YOUR_JWT_TOKEN"

# Upload a document

curl -X POST "http://localhost:5000/api/buckets/my-bucket/documents" \ -H "Content-Type: multipart/form-data" \ -H "Authorization: Bearer YOUR_JWT_TOKEN" \ -F "file=@/path/to/document.pdf" \ -F "key=reports/2023/report.pdf"

# Search for documents

curl -X GET "http://localhost:5000/api/buckets/my-bucket/documents/search?query=quarterly%20report" \ -H "Authorization: Bearer YOUR_JWT_TOKEN" ```
```

Using Python

```python import requests

API_URL = "http://localhost:5000/api" TOKEN = "YOUR_JWT_TOKEN"

# List documents in a bucket

response = requests.get( f"{API_URL}/buckets/my-bucket/documents", params={"prefix": "reports/"}, headers={"Authorization": f"Bearer {TOKEN}"} ) print(response.json())
```

Read a document

response = requests.get(f"{API_URL}/buckets/my-bucket/documents/reports/2023/report.pdf", headers={"Authorization": f"Bearer {TOKEN}"}) print(response.json()) ```

9. Testing API Endpoints

1. **Unit Testing**
    - Test API endpoints with mock S3
    - Verify request validation
    - Check response formats
    - **Integration Testing**
    - Test with actual S3 buckets (test environment)
    - Verify end-to-end workflows
    - Test error scenarios
    - **Performance Testing**
    - Measure API response times
    - Test with large documents
    - Evaluate concurrent request handling

Conclusion

This implementation plan outlines the steps to create a robust and feature-rich S3 Document Reader/Writer API service. The API design provides a flexible interface for managing documents in S3 buckets, with support for various document formats and operations. The modular design allows for easy extension and maintenance, while the comprehensive API endpoints provide powerful document management capabilities for an MVP implementation.