



[Pre-requisite] Enable foundation model access in Amazon Bedrock

[Pre-requisite] Configuring the front-end application

► Playground

▼ Use cases

► Building a RAG pipeline

▼ Document extraction and summarization

► Intelligent document processing with Generative AI

▼ Scaling with serverless workflows

Building the workflow

Verifying the workflow execution

High level Code Walkthrough

Scheduling using Amazon

EventBridge Scheduler

▼ AWS account access

[Open AWS console \(us-west-2\)](#)

[Get AWS CLI credentials](#)

Exit event

[Event dashboard](#) > [Use cases](#) > [Document extraction and summarization](#) > [Scaling with serverless workflow](#)

High level Code Walkthrough

The document extraction and document summarization Lambda functions both uses [Rhubarb Library](#) to perform document extraction and summarization. **Rhubarb** is a light-weight Python framework that makes it easy to build document understand application using Large Language Models(LLMs) and embedding models. Rhubarb is created from the ground-up to work with Amazon Bedrock and [Anthropic Claude 3 model](#) family of LLMs which can understand both text and image. We can use PDFs that have images without using a separate model that can understand image.

Lambda function code for document Extraction

```

1  import json
2  import boto3
3  from rhubarb import DocAnalysis, LanguageModels
4  from botocore.exceptions import ClientError
5  import os
6  import uuid
7
8  # Fetch the AppSync API key from environment variable
9  APP_SYNC_API_KEY = os.environ.get('API_KEY')
10 API_ENDPOINT = os.environ.get('API_ENDPOINT')
11
12 def lambda_handler(event, context):
13     # Initialize boto3 session
14     session = boto3.Session()
```



```
15
16     s3 = boto3.client('s3')
17
```

© 2008 - 2025, Amazon Web Services, Inc. or its affiliates. All rights reserved. [Privacy policy](#) [Terms of use](#) [Cookie preferences](#)

```
20     "type": "object",
21     "properties": {
22         "infrastructure_cost": {
23             "description": "Cost related to infrastructure hosting",
24             "type": "string"
25         },
26         "development_cost": {
27             "description": "Cost related to application development",
28             "type": "string"
29         },
30         "maintenance_cost": {
31             "description": "Cost related to maintaining the application",
32             "type": "string"
33         },
34         "case_study_1_overview": {
35             "description": "Overview of the first case study",
36             "type": "string"
37         },
38         "cost_calculation": {
39             "description": "Cost calculation details",
40             "type": "string"
41         }
42     },
43     "required": [
44         "infrastructure_cost",
45         "development_cost",
46         "maintenance_cost",
47         "case_study_1_overview",
48         "cost_calculation"
49     ]
50 }
51
```

```
52 # Get the file key from the event
53 file_key = event['file_key']
54
55 # Assume bucket name is stored in an environment variable
56 bucket_name = os.environ['BUCKET_NAME']
57
58 file_path = f"/tmp/{file_key.split('/')[-1]}"
59 s3.download_file(bucket_name, file_key, file_path)
60
61 try:
62     da = DocAnalysis(file_path=file_path, boto3_session=session, modelId=LanguageModels)
63     resp = da.run(message="Give me the output based on the provided schema.", output_schema=output_schema)
64
65     id_uuid = str(uuid.uuid4())
66
67     # Prepare the item for GraphQL mutation
68     item = {
69         'id': id_uuid,
70         'full_json': json.dumps(resp)
71     }
72
73     # Process extracted fields and add them to the item
74     if 'output' in resp:
75         output = resp['output']
76         item.update({
77             'infrastructure_cost': output.get('infrastructure_cost', ''),
78             'development_cost': output.get('development_cost', ''),
79             'maintenance_cost': output.get('maintenance_cost', ''),
80             'case_study_1_overview': output.get('case_study_1_overview', ''),
81             'cost_calculation': output.get('cost_calculation', '')
82         })
83
84 except Exception as e:
85     return {
86         'statusCode': 500,
87         'body': json.dumps(f"Error processing file: {str(e)}")
88     }
```

```
}
```

1. Walk through the code to understand different functionalities.
2. The JSON schema defines the structure of the data to be extracted from the document. From our PDF doc, we wanted to extract data related to `infrastructure_cost`, `development_cost`, and `maintenance_cost`. So, we created a schema with those properties.

```
# JSON schema for the document analysis
{
  "type": "object",
  "properties": {
    "infrastructure_cost": {
      "description": "Cost related to infrastructure hosting",
      "type": "string"
    },
    "development_cost": {
      "description": "Cost related to application development",
      "type": "string"
    },
    "maintenance_cost": {
      "description": "Cost related to maintaining the application",
      "type": "string"
    },
    "case_study_1_overview": {
      "description": "Overview of the first case study",
      "type": "string"
    },
    "cost_calculation": {
      "description": "Cost calculation details",
      "type": "string"
    }
  }
},
```

```
"required": [  
    "infrastructure_cost",  
    "development_cost",  
    "maintenance_cost",  
    "case_study_1_overview",  
    "cost_calculation"  
]  
}
```

3. The Rhubarb Library interacts with Amazon Bedrock to leverage large language models (LLMs) like Claude Haiku. When the document is processed using the `DocAnalysis` class, the Rhubarb Library uses the AWS SDK (boto3) to send API requests to Bedrock, invoking the Claude Haiku model to extract data according to the defined schema.

```
1      da = DocAnalysis(file_path=file_path, boto3_session=session, modelId=LanguageModelId.LLM_HAiku_v1_0_0)
2      resp = da.run(message="Give me the output based on the provided schema.", output_schema=schema)
```

Alternative Approach: Implementing without Rhubarb Library

If not using the **Rhubarb** Library, you can achieve similar functionality by directly interacting with Amazon Bedrock and implementing custom document processing logic. This approach involves parsing documents using libraries like **PyPDF2**, defining your own schema for information extraction, and creating custom prompts for the Claude V3 model. You will need to use the AWS SDK (boto3) to integrate with Amazon Bedrock, make API calls to the Claude V3 model, and implement your own logic for parsing responses and structuring extracted data.

Handling Dynamic Schemas for Each document


Currently, the schema is hardcoded into the Lambda function for simplicity, as both documents use the same structure. However, in real-world scenarios where the schema might vary for each document, a more flexible approach is needed.

One pattern we could adopt is externalizing the schema into the manifest.json file alongside the document's location. This way, each document can be associated with its own unique schema, allowing the Lambda function to dynamically read and apply the correct schema during processing.

Example manifest.json structure:

```
{
  "file_key": "serverless-document.pdf",
  "schema": {
    "type": "object",
    "properties": {
      "infrastructure_cost": {
        "description": "Cost related to infrastructure hosting",
        "type": "string"
      },
      "development_cost": {
        "description": "Cost related to application development",
        "type": "string"
      }
    },
    "required": [
      "infrastructure_cost",
      "development_cost"
    ]
  }
}
```

Typical Workflow in Production

In the previous module, we triggered the workflow manually. However, in real-world scenarios where hundreds of documents are processed, it's common to run workflows in batch processing mode. This is typically done using [EventBridge Scheduler](#) , which triggers the workflow at scheduled intervals rather than on each individual file upload.

[Previous](#)[Next](#)