Understand the Lambda function Code

This point breaks down the Lambda function code that powers the AWS Receipt Processing System, explaining each major component and its purpose.

Code Structure Overview

The Lambda function is organized into four main components:

1. **Lambda Handler**: Entry point that coordinates the entire workflow
2. **Textract Processing**: Extracts structured data from receipt images
3. **DynamoDB Storage**: Saves the processed data to the database
4. **Email Notification**: Sends formatted results via email

Lambda Handler Function

```
def lambda_handler(event, context):
    try:
        # Get the S3 bucket and key from the event
        bucket = event['Records'][0]['s3']['bucket']['name']
        # URL decode the key to handle spaces and special characters
        key = urllib.parse.unquote_plus(event['Records'][0]['s3']['object']['key'])

        # Verify the object exists before proceeding
        s3.head_object(Bucket=bucket, Key=key)

        # Step 1: Process receipt with Textract
        receipt_data = process_receipt_with_textract(bucket, key)
```

```
        # Step 2: Store results in DynamoDB
        store_receipt_in_dynamodb(receipt_data, bucket, key)

        # Step 3: Send email notification
        send_email_notification(receipt_data)

        return {
            'statusCode': 200,
            'body': json.dumps('Receipt processed successfully!')
        }
    except Exception as e:
        print(f"Error processing receipt: {str(e)}")
        return {
            'statusCode': 500,
            'body': json.dumps(f'Error: {str(e)}')
        }
```

What it does:

- Acts as the orchestrator for the entire process
- Extracts information about which file was uploaded from the S3 event
- URL-decodes the file path to handle special characters
- Verifies the file exists before attempting processing
- Calls the specialized functions for each step of the process
- Handles errors gracefully with detailed logging

Textract Processing Function

```
def process_receipt_with_textract(bucket, key):
    # Call Textract to analyze the receipt
    response = textract.analyze_expense(
        Document={
```

```
            'S3Object': {
                'Bucket': bucket,
                'Name': key
            }
        }
    )

    # Generate a unique ID for this receipt
    receipt_id = str(uuid.uuid4())

    # Initialize receipt data with default values
    receipt_data = {
        'receipt_id': receipt_id,
        'date': datetime.now().strftime('%Y-%m-%d'),
        'vendor': 'Unknown',
        'total': '0.00',
        'items': [],
        's3_path': f"s3://{bucket}/{key}"
    }

    # Process summary fields (TOTAL, DATE, VENDOR)
    if 'ExpenseDocuments' in response and response['ExpenseDocuments']:
        expense_doc = response['ExpenseDocuments'][0]

        # Extract key fields like vendor, date, total
        if 'SummaryFields' in expense_doc:
            for field in expense_doc['SummaryFields']:
                field_type = field.get('Type', {}).get('Text', '')
                value = field.get('ValueDetection', {}).get('Text', '')

                # Map fields to our data structure
                if field_type == 'TOTAL':
                    receipt_data['total'] = value
                elif field_type == 'INVOICE_RECEIPT_DATE':
                    receipt_data['date'] = value
                elif field_type == 'VENDOR_NAME':
```

```
                receipt_data['vendor'] = value

        # Extract line items from the receipt
        if 'LineItemGroups' in expense_doc:
            for group in expense_doc['LineItemGroups']:
                if 'LineItems' in group:
                    for line_item in group['LineItems']:
                        item = {}
                        for field in line_item.get('LineItemExpenseFields', []):
                            field_type = field.get('Type', {}).get('Text', '')
                            value = field.get('ValueDetection', {}).get('Text', '')

                            if field_type == 'ITEM':
                                item['name'] = value
                            elif field_type == 'PRICE':
                                item['price'] = value
                            elif field_type == 'QUANTITY':
                                item['quantity'] = value

                        # Add to items list if we have a name
                        if 'name' in item:
                            receipt_data['items'].append(item)

    return receipt_data
```

What it does:

- Uses Amazon Textract's specialized `analyze_expense` API
- Creates a unique ID for the receipt
- Sets up default values for all expected fields
- Extracts summary information (vendor, date, total amount)
- Extracts individual line items with their quantities and prices
- Returns structured data in a consistent format

Key insights:

- Textract understands the structure of receipts, not just the text
- The function handles missing data gracefully with default values
- The unique ID ensures each receipt can be tracked independently

DynamoDB Storage Function

```
def store_receipt_in_dynamodb(receipt_data, bucket, key):
    table = dynamodb.Table(DYNAMODB_TABLE)

    # Convert items to a format DynamoDB can store
    items_for_db = []
    for item in receipt_data['items']:
        items_for_db.append({
            'name': item.get('name', 'Unknown Item'),
            'price': item.get('price', '0.00'),
            'quantity': item.get('quantity', '1')
        })

    # Create item to insert
    db_item = {
        'receipt_id': receipt_data['receipt_id'],
        'date': receipt_data['date'],
        'vendor': receipt_data['vendor'],
        'total': receipt_data['total'],
        'items': items_for_db,
        's3_path': receipt_data['s3_path'],
        'processed_timestamp': datetime.now().isoformat()
    }

    # Insert into DynamoDB
    table.put_item(Item=db_item)
```

What it does:

- Connects to the DynamoDB table

- Formats the receipt data for database storage
- Adds a processing timestamp for tracking
- Stores all receipt information as a single item
- Includes the S3 path to link back to the original document

Key insights:

- The structured format makes it easy to query receipts later
- The timestamp records when processing occurred
- The S3 path allows you to access the original receipt if needed

Email Notification Function

```
def send_email_notification(receipt_data):
    # Format items for email
    items_html = ""
    for item in receipt_data['items']:
        name = item.get('name', 'Unknown Item')
        price = item.get('price', 'N/A')
        quantity = item.get('quantity', '1')
        items_html += f"<li>{name} - ${price} x {quantity}</li>"

    # Create email body
    html_body = f"""
    <html>
    <body>
        <h2>Receipt Processing Notification</h2>
        <p><strong>Receipt ID:</strong> {receipt_data['receipt_id']}</p>
        <p><strong>Vendor:</strong> {receipt_data['vendor']}</p>
        <p><strong>Date:</strong> {receipt_data['date']}</p>
        <p><strong>Total Amount:</strong> ${receipt_data['total']}</p>
        <p><strong>S3 Location:</strong> {receipt_data['s3_path']}</p>

        <h3>Items:</h3>
```

```
        <ul>
            {items_html}
        </ul>

        <p>The receipt has been processed and stored in DynamoDB.</p>
    </body>
    </html>
    """

    # Send email using SES
    ses.send_email(
        Source=SES_SENDER_EMAIL,
        Destination={
            'ToAddresses': [SES_RECIPIENT_EMAIL]
        },
        Message={
            'Subject': {
                'Data': f"Receipt Processed: {receipt_data['vendor']} - ${receipt_dat
a['total']}"
            },
            'Body': {
                'Html': {
                    'Data': html_body
                }
            }
        }
    )
```

What it does:

- Creates a formatted HTML email with receipt details
- Includes a list of all extracted line items
- Uses Amazon SES to send the email
- Creates a descriptive subject line with vendor and total
- Includes the S3 path to access the original document

Key insights:

- HTML formatting makes the email easy to read
- Including line items provides complete information
- The receipt ID and S3 path enable tracking and retrieval

Error Handling and Logging

Throughout the code, you'll notice:

- Try-except blocks that catch and log errors
- Detailed log messages at each processing step
- Default values to handle missing data gracefully
- Continuation of execution when non-critical parts fail (e.g., email)

This robust error handling ensures the system can process imperfect receipts and recover from temporary issues without manual intervention.

Environment Variables

The code uses these environment variables for configuration:

- `DYNAMODB_TABLE`: The name of the DynamoDB table to store receipts
- `SES_SENDER_EMAIL`: The verified email address to send notifications from
- `SES_RECIPIENT_EMAIL`: The email address to receive notifications

Using environment variables allows you to change these settings without modifying the code.