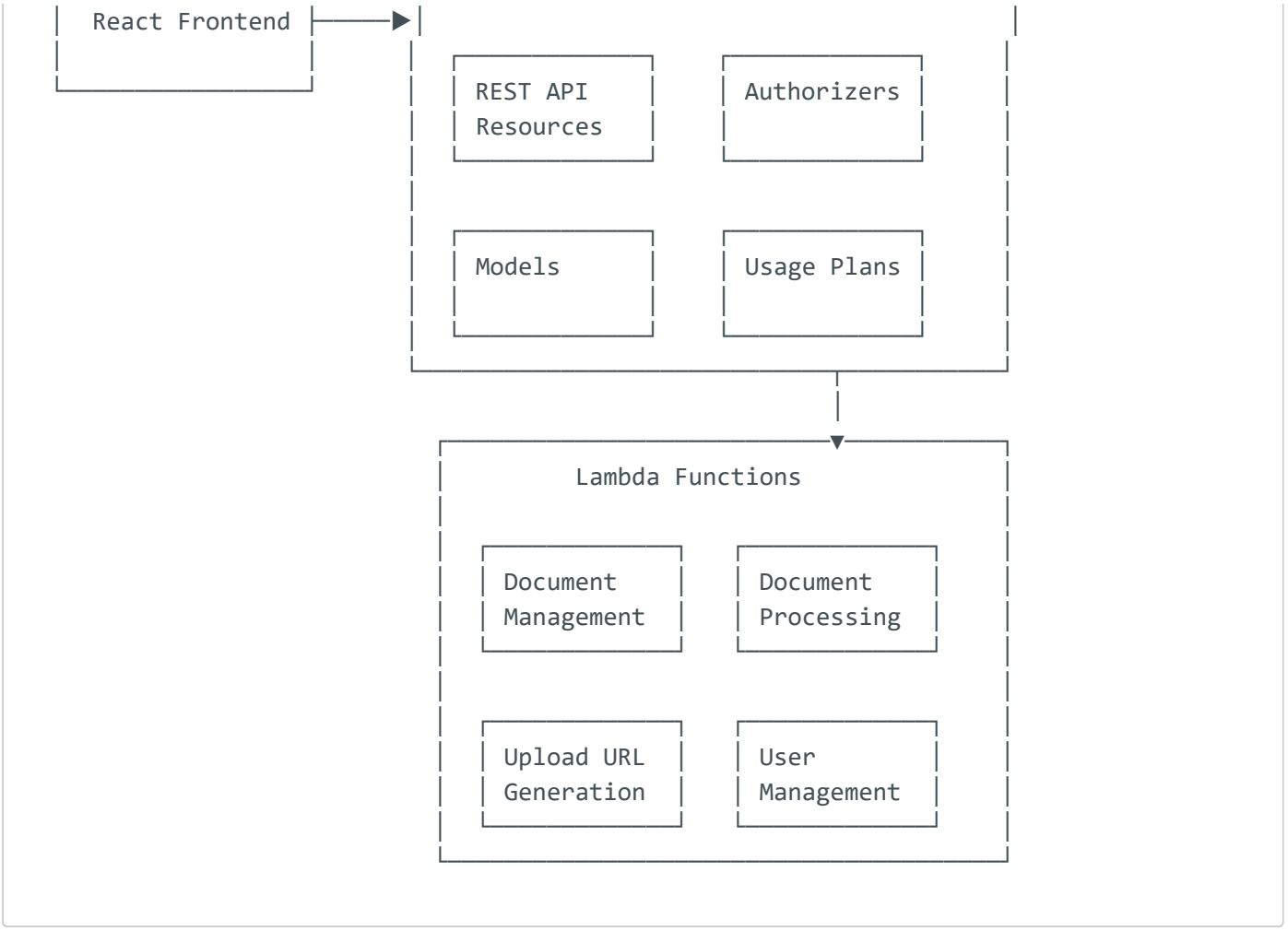
API Architecture

This document outlines the API architecture of the Document Processing Accelerator, detailing the endpoints, authentication flow, error handling, and integration patterns.

API Gateway Architecture

The Document Processing Accelerator uses AWS API Gateway to expose a RESTful API that serves as the communication layer between the frontend application and backend services.



API Endpoints

The API is organized into resource-focused endpoints:

Document Management Endpoints

Method	Endpoint	Description	Auth Required
GET	/documents	List all documents for the current user	Yes
GET	/documents/{id}	Get a specific document by ID	Yes
POST	/documents	Create a new document	Yes
DELETE	/documents/{id}	Delete a document by ID	Yes
POST	/documents/upload-url	Generate a pre-signed URL for file upload	Yes

Method	Endpoint	Description	Auth Required
POST	/documents/{id}/process	Process a document	Yes

Swagger Documentation Endpoints

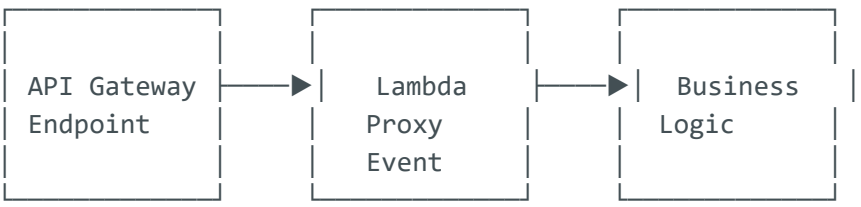
Method	Endpoint	Description	Auth Required
GET	/swagger	Get the OpenAPI specification	No
GET	/swagger/ui	Access the Swagger UI documentation	No

API Integration Pattern

The API Gateway implements several integration patterns for different backend operations:

Lambda Integration

The primary integration pattern is Lambda Proxy integration:



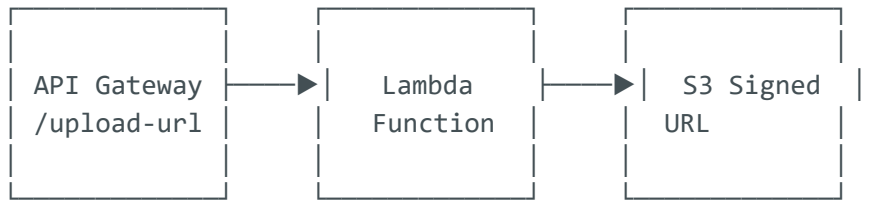
For Lambda Proxy integrations, the entire HTTP request is passed to the Lambda function, including:

- HTTP method
- Headers
- Query string parameters
- Request body
- Path parameters

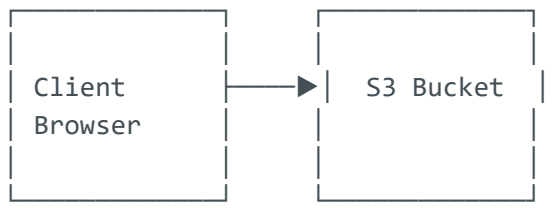
S3 Integration

For document uploads, we use a two-step process:

1. Generate pre-signed URL:



2. Client uploads directly to S3:

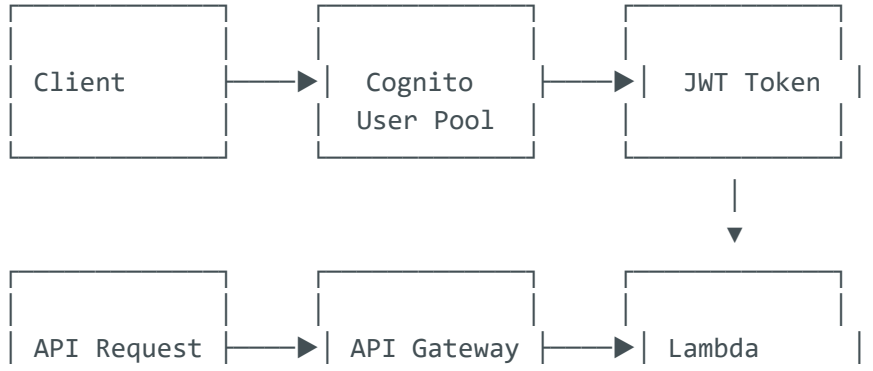


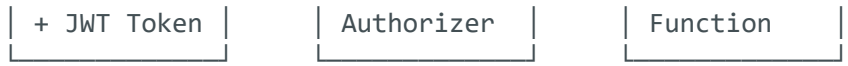
This pattern minimizes API Gateway data transfer costs and provides efficient large file uploads.

Authentication and Authorization

The API uses Cognito for authentication and JWT tokens for authorization.

Authentication Flow





Authorization Implementation

API Gateway Authorizer

The Cognito User Pool is configured as an authorizer for the API Gateway:

```
# serverless.yml excerpt
authorizers:
  cognito:
    type: COGNITO_USER_POOLS
    providerARNs:
      - ${self:custom.userPoolArn}
```

Token Validation

The JWT token validation validates:

- Token signature using Cognito's public keys
- Token expiration
- Audience claim (client ID)
- Issuer claim (Cognito User Pool URL)

API Models and Type Safety

The API is built with TypeScript for end-to-end type safety.

Request Validation

Request validation happens at multiple levels:

1. **API Gateway Model Validation**: JSON Schema-based validation for request bodies
2. **TypeScript Type Checking**: Strong typing in Lambda functions
3. **Runtime Validation**: Additional runtime validation using validation libraries

Example model:

```
// Document model
export interface Document {
  id: string;
  userId: string;
  title: string;
  documentType: DocumentType;
  status: DocumentStatus;
  metadata?: Record<string, any>;
  s3Key?: string;
  createdAt: string;
  updatedAt: string;
}

// Input model for document creation
export interface DocumentInput {
  title: string;
  documentType: DocumentType;
  description?: string;
}
```

CORS Configuration

Cross-Origin Resource Sharing is configured at the API Gateway level:

```
# serverless.yml excerpt
cors:
  origin: ${env:FRONTEND_URL, '*'}
  headers:
    - Content-Type
    - X-Amz-Date
    - Authorization
    - X-Api-Key
    - X-Amz-Security-Token
    - X-User-Id
  allowCredentials: true
```

Error Handling

The API implements a standardized error response format:

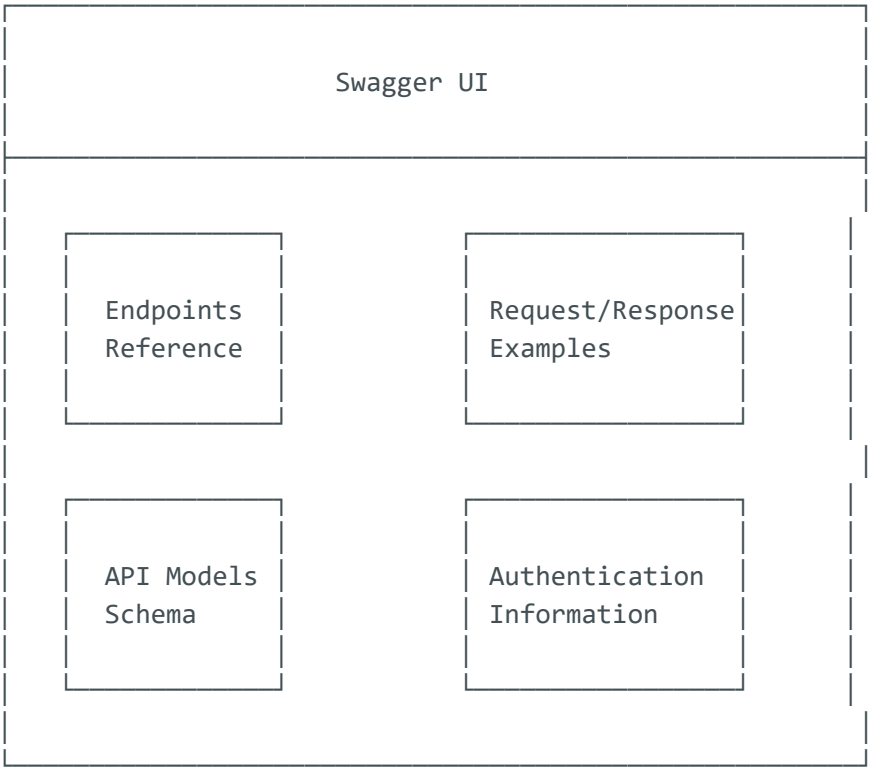
```
interface ErrorResponse {
  error: {
    code: string;
    message: string;
    details?: any;
  }
}
```

HTTP status codes are used appropriately:

- 400 Bad Request: Invalid input
- 401 Unauthorized: Authentication required
- 403 Forbidden: Insufficient permissions
- 404 Not Found: Resource not found
- 500 Internal Server Error: Unexpected server error

API Documentation

The API is documented using Swagger/OpenAPI:



Swagger Integration

The API uses two dedicated Lambda functions to serve the OpenAPI documentation:

1. `serveSwaggerJson`: Returns the OpenAPI specification as JSON
2. `serveSwaggerUI`: Serves the Swagger UI HTML interface

API Versioning

The API version is managed through the URL path:

```
https://api-endpoint.com/dev/documents  (development stage)
https://api-endpoint.com/prod/documents (production stage)
```

Future API versions can be implemented as:

```
https://api-endpoint.com/v2/documents (version 2)
```

Rate Limiting and Quotas

API Gateway usage plans provide rate limiting:

- Default limit: 1000 requests per minute
- Burst limit: 2000 requests

Custom throttling limits can be applied per client using API keys.

Implementation Details

The API is implemented using TypeScript Lambda functions:

```
// Example Lambda handler (documents/get.ts)
export const handler = async (event: APIGatewayProxyEvent):
Promise<APIGatewayProxyResult> => {
  try {
    // Extract user ID from JWT token
    const userId = getUserIdFromToken(event);

    // Query DynamoDB for documents
```

```
    const documents = await documentService.getDocumentsByUserId(userId);

    // Return successful response
    return {
      statusCode: 200,
      headers: corsHeaders,
      body: JSON.stringify(documents)
    };
  } catch (error) {
    // Handle errors
    return handleError(error);
  }
};
```

API Client Integration

The frontend includes a strongly-typed API client for interacting with the backend:

```
// apiClient.ts
import { authService } from './authServiceProvider';

export const apiClient = {
  async getDocuments(): Promise<Document[]> {
    const token = await authService.getToken();
    return this.get('/documents', token);
  },

  async createDocument(input: DocumentInput): Promise<Document> {
    const token = await authService.getToken();
    return this.post('/documents', input, token);
  },

  // Base methods
  private async get(endpoint: string, token: string): Promise<any> {
    return fetch(`${API_BASE_URL}${endpoint}`, {
      headers: {
        'Authorization': `Bearer ${token}`,
        'Content-Type': 'application/json'
      }
    }).then(this.handleResponse);
  },

  private async post(endpoint: string, data: any, token: string): Promise<any> {
    return fetch(`${API_BASE_URL}${endpoint}`, {
      method: 'POST',
      headers: {
        'Authorization': `Bearer ${token}`,
        'Content-Type': 'application/json'
      },
      body: JSON.stringify(data)
    }).then(this.handleResponse);
```

```
  }
};
```

Future API Enhancements

Planned enhancements for the API include:

1. **GraphQL Support**: Adding GraphQL endpoints for more flexible data fetching
2. **WebSocket API**: Real-time notifications for document processing status
3. **API Caching**: Response caching for improved performance
4. **Regional Deployment**: Multi-region API deployment for lower latency