AWS Cloud Computing

Project Report

DocAI Organizer

Organizing documents with AI

Author: Henri Cerio-Cain

Date: December 18th, 2025

1. Project Introduction

1.1 Overview

DocAI Organizer is a straightforward, yet effective cloud application designed to assist with a task almost everyone has done in their lives: document organization. This often tedious task was something I identified as ideal for automation using AI, by taking a task that normally requires hours of work and reducing it to seconds. Users create custom folder structures, upload their documents, AI automatically classifies and organizes files based on content analysis, and download the new, organized folder structure with all files present.

1.2 Features

At a high level, users automate document organization by creating the desired file structure, uploading their documents, and pressing "Organize." Google Gemini 2.5 Flash model then analyzes each document's content and automatically places it into the appropriate folder based on the structure provided. Once complete, users can preview files in the browser, make any needed adjustments, and download everything as a ZIP file that maintains their folder structure.

To handle different needs, users can create as many folder structures as they want through something called “conversations”. Each conversation acts like a separate space with its own folder tree, saving the progress of each conversation with the ability to also search for them through their name, which is also customizable. All of these are stored in the users’ accounts, so with a simple login, users can access everything securely.

1.3 Target Users

DocAI Organizer is targeted for a large set of users. Students, office workers, or even just someone who wants to tidy up their computer can all benefit from using this product. Built on the web allows for all sorts of devices to be able to access DocAI Organizer ubiquitously.

1.4 Performance Requirements

With an application surrounding automating the organization of PDFs through AI, it’s clear that this process must be faster than manual organization for it to be feasible for real-world use. Selecting Google Gemini’s 2.5 Flash model allows for speedy yet accurate PDF organization, which in practise, allowed for sub-30-second completion of variable file counts. During this process of organizing documents, another sub-requirement is that the website remains responsive to ensure a professional user experience. While the application architecture is built on AWS Lambda, which is notorious for cold starts, the system is aimed

to handle cold starts gracefully with automatic scaling for variable user counts, with a target response time under 3 seconds, with the AI organization being the longest operation.

2. AWS Service Stack

DocAI Organizer was built with four main AWS services: Lambda, S3, API Gateway and DynamoDB. This technology stack was meticulously chosen to leverage the application's philosophy with services that minimize cost, while maintaining a professional user experience.

2.1 Compute Service

AWS Lambda was used for the backend of the cloud application, executing code based on the HTTP requests made from the API Gateway. It handles all the endpoints of the app, from user registration to deleting a file, and it is the bridge that handles S3 and DynamoDB operations made by the user. Given the nature of the app being a simple automation task, Lambda made the most sense out of all other options, due to the parallels between their use cases.

Despite it working almost perfectly, the nature of AWS Lambda means cold starts, which means slightly slow responses, for which EC2 was heavily considered. While EC2 would give DocAI Organizer a much faster response time, many downsides would come with choosing it over Lambda. EC2's cost structure is time-based, and not when used, compared to Lambda. EC2 also comes with server maintenance, security updates and scaling configuration, and with the nature of document organization being a sporadic activity, and not constant, EC2's benefits are negligible in comparison to the downsides.

2.2 Storage Service

Amazon Simple Storage Service (S3) was used to store users ' PDFs and documents on the cloud. S3 was the smartest approach, I feel, to store documents. Elastic File System (EFS), when compared to S3, came with higher rates. DynamoDB binary storage has file size constraints, which conflict with the nature of the cloud app, as PDF's vary in size. On top of other alternatives not making sense, S3 also came with future-proof scalability. Not only does it allow for unlimited size scalability for users to upload as many documents as they want, S3 also allows for direct file upload via URLs and is extremely cost-effective for large files, which PDFs usually fall under. The storage cost is only $0.023 per GB per month, which means even storing thousands of PDFs costs almost nothing [5].

2.3 Network Delivery

API Gateway was used to route HTTP requests made to the AWS Lambda. It effectively listens for requests made and based on the request, routes these requests to the lambda function. I primarily went with API Gateway as I have used it in the past, so I was already familiar with

configuring it. In union to experience, API Gateway also works in harmony with AWS Lambda. API Gateway handles request validation, ensuring that all required parameters are present before sending data over to Lambda. This prevents malformed requests immediately, which saves the cost of running Lambda.

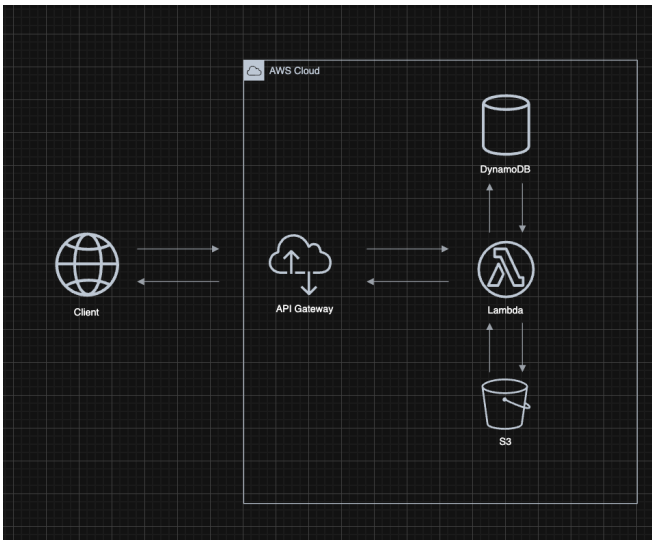
2.4 Database Service

Amazon DynamoDB was used as the database management system. It's used for storing user credentials and history. It was the most attractive option out of all the DBMS Amazon offers, from its autoscaling and serverless pricing, which is perfect given the nature of this application. The architecture of the database consists of a user's table, which stores a user's email, hash and metadata, and a conversations table that stores all the folder structures. These two tables use composite and partition keys, which allow for efficient queries used in Lambda. DynamoDB's autoscaling enables variable usage without manual interception, and the on-demand pricing complements the nature of the unpredictability of usage upon release.

A heavily considered alternative was Amazon RDS in union with EC2; however, it was ultimately deemed not fit based on the application's needs. I wanted to design DocAI Organizer as simply as possible, and with the main function of the app not needing complex database operations or managing backups, DynamoDB's key-value structure allowed for my intended function, as simple as possible.

3. Application Architecture

3.1 Architecture diagram



3.2 Programming Languages

HTML, CSS and JavaScript were the programming languages used to create this cloud application. HTML, CSS and Bootstrap were used for the front-end UI for the website's pages, and JavaScript was used to create the front-end and backend functionality. I'm familiar with JavaScript and server-side scripting through class work and even in my personal projects, so wiring up promises in the front-end JavaScript file and creating the endpoints in the lambda file that manipulate DynamoDB and S3 came naturally. Notably, the Node.js lambda backend uses external libraries like brcypt for hashing passwords and to create JWT Tokens for user authentication; jsonwebtoken was used. To download files following the tree structure, the front end utilizes the JSZip library.

The lambda function and the front-end all required code, but all the other portions of the application, like creating a DB was all done via AWS's console.

3.3 Component Flow

The four components, Lambda, Amazon Simple Storage Service, API Gateway and DynamoDB, all work in harmony to fully create the application and keep it running. Firstly, the API gateway listens for HTTP requests from the browser. Upon a request, the API gateway sends an event object to Lambda. Based on the endpoint, Lambda queries DynamoDB for endpoints that require updating a user's "conversation," and/or S3 for endpoints that work with the actual PDFs stored on the cloud, such as uploading or deleting a file.

3.3.1 Organization Logic

To organize files given a folder structure, the application follows the following process:

1. User uploads a document through the dropbox, which calls a POST endpoint that passes the document's file name.
2. The lambda returns a generated a pre-signed URL to the user's S3 folder to upload the file, which the frontend uses for a direct upload to S3.
3. Upon the organize button being pressed, the browser calls a POST endpoint specifically for classification, passing the S3 key.
4. The Lambda function validates the S3 key, then calls DynamoDB to parse the tree in JSON format, extracting all folder paths, and placing them into an array for later validation.
5. Lambda then calls S3 to retrieve all the PDF's and translate them into a buffer, to which the Gemini API is called with a prompt that organizes each file into a folder in JSON format.
6. Lambda parses Gemini's JSON response and validates each output through the earlier array to catch for hallucinations or poor outputs.
7. Based on the validated response, Lambda calls CopyObject and DeleteObject in S3 to essentially move a file to its correct position.
8. Lambda finalizes by returning the assigned path to the frontend, which refreshes the file structure to show the now updated locations of files.

3.3.2 Export Logic

To download a tree and export it as a zip file, DocAI Organizer logic starts through a GET endpoint that returns the metadata for all files. With this metadata, the browser then sends a POST request to get each pre-signed S3 URL for direct downloading. Each file is downloaded using FetchAPI, while collecting the bytes that are added to a JSZip instance with the folder structure extracted from the S3 key path. After all files are added, JSZip's generateAsync() function is called, which downloads the documents using a temporary download link.

3.4 Deployment

Since Lambda used external libraries, externally downloading the files and uploading the files through the AWS console with node_modules was required. The execution time was set for 30 seconds to account for the longer execution times made by the backend API call for Gemini, with memory configured to 512MB to provide enough power for PFD processing and API calls.

Regarding storage, S3 has default encryption, and the bucket was set to private. CORS was configured to allow PUT requests from any origin (which would change to the domain upon release) for direct uploads.

To deploy DocAI Orgnaizer outside of localhost, the website can be hosted through another S3 bucket that would be configured for web hosting.

4. Security

With an application that works with user documents, security was a major consideration throughout development. Given the scope of the application, security was focused on three main areas: protecting user credentials, ensuring authenticated users can only access their own data, and allowing access to stored documents.

4.1 Database Security

Through my server-side scripting course, I learned a lot about how servers work with databases that store sensitive data like passwords. Here, I learned about the system of storing hashes of passwords, comparing a user's input password, and using the hash algorithms to see if the input password matches. I implemented this same system with DocAI Organizer, using the bcrypt hashing algorithm.

Upon user registration, the user's actual password is never actually stored, as we store their hashed password with a salt rounds value of 12. The number 12 was chosen to balance protection and feasibility. This value is big enough for security against attackers, but also low enough to which hashes are quick. On top of the strong minimum requirements of a valid password being one uppercase, one number and one special character, registration is secure, but also feasible.

When the user logs into an account, Lambda retrieves the user record from DynamoDB using the email index, and calls bcrypts native method to compare the input password with the stored hash. If the credentials match, users are navigated accordingly, and if they fail, a vague toast regarding invalid credentials is displayed to provide enough information without overexposing information.

4.2 Login Security

Since it is built on top of AWS Lambda, a stateless resource, finding out how to implement log-in functionality while keeping it secure was an important task. To achieve this, JSON Web Tokens were used to simulate server-side sessions to keep track of who's logged in, while keeping security tight.

When users log in, a JWT is created that keeps track of the user's id, a secret, and the expiry of the token. This secret is stored on the server side through environmental variables to validate potentially fraudulent tokens. Once this is returned to the user, the browser stores it in local storage, and with each subsequent request, this JWT is attached to keep users authenticated in their accounts.

To decode a JWT on the server, we use the JWT verify function that takes a token and our secret, from which the secret and expiration are extracted. Depending on whether the secret matches and the expiration is still valid, the request either passes the check, or a redirection to the login with a 401 Unauthorized.

This strategy is effective in that it checks for a valid JWT token with every single request, so not once is there a potential disconnect. It also protects a user's account from tampering, as even if an attacker were to decode their token and change the user id, the backend verification of the secret fails immediately. With a 7-day expiration of each JWT, users can gracefully use DocAI Organizer without needing constant authentication, all while potentially compromised tokens that are past die.

4.3 Storage Security

Now that users can be authenticated securely, another big security task was finding a secure way to store the documents and PDFs of users. Thankfully, S3 did a lot of the work natively, and it was just a case of wiring them up together.

The S3 bucket was configured as private with block all access enabled, which allows for no-one being able to access the contents directly. With pre-signed URLs, temporary URLs that work for a couple of minutes for that user's specific folder in which their data is stored, this allowed for keeping the bucket private, however, leaving openings for users to still be able to interact with it for actions such as uploading or deleting files. These URLS are generated only after validating that the S3 key belongs to the authenticated user's conversation. By default, these URLs expire after 5 minutes, ensuring we follow the minimum privilege with the minimum time security philosophy.

4.4 Security Vulnerabilities

Despite the current security measures, there is still one big security issue that should be solved prior to public use. Currently, any user can use DocAI Organizer as many times as they want. An attacker could exploit this gap to spam the endpoint in which the API call to Gemini is made, which would ramp up costs significantly or even DDoS the application entirely. Rate limiting is an industry standard for AI applications such as DocAI Organizer that explicitly prevents this issue. Further research is required to implement such a technique with our current architecture.

5. Cost Metrics

5.1 Upfront Cost

The upfront cost of DocAI Organizer is $0. To create the cloud application, the free trail AWS Learners Labs was used, which contains $50 in free credits. Each service used (Lambda, API Gateway, S3 and DynamoDB) uses the $50 credit before external payment, alongside any cost of the API call for the backend AI model Gemini Flash 2.5, which would increase the cost of DocAI Organizer.

Using Lambda for the backend of the code allows for zero servers to buy and use to develop the code, which means zero upfront costs to create the product. Unlike other services that use servers, which scale with time rather than usage, Lambda offers the first one million requests for free with zero other fees attached [1]. Given that less than one million requests were made to develop DocAI Organizer, no fees were charged upon the Learner's Labs credits.

Using API Gateway to route and listen for HTTP requests is extremely affordable. With their free tier, they allow for one million API calls for free each month [2]. Given that fewer than one million requests were made for the development, no fees were incurred on the Learner's Lab free credits.

Unlike the other services used, DynamoDB doesn't offer a certain number of requests for free; however, the cost for the number of requests made for development was negligible. With the cost for 100 reads being $0.000025, and the number of reads being used for development, this number is extremely close to nothing [4]. For simplicity, the cost of using DynamoDB will be rounded down to $0.

S3 follows the same sort of format as DynamoDB, in the sense that no free uses are offered [5]. S3 storage costs $0.023 per GB per month for the first 50TB, with additional charges of $0.005 per 1000 PUT requests and $0.0004 per 1000 GET requests[5]. Given the size and number of files used to test DocAI Organizer, the cost amounts to pennies towards the Learner's Lab free credits, in which this value is also rounded to $0.

Lastly, Gemini's 2.5 Flash API offers a free tier that was used for development, that allow for 1500 requests per day, which was more than enough for development and testing [6].

5.2 Ongoing Costs

For the ongoing cost, we will assume that there will be 1000 active users monthly who use 2 conversations with 20 PDFs with 2 API calls for Gemini per month. By these metrics, we assume the worst-case scenario for each service to ensure a buffer is present.

Lambda Compute at the worst case, would receive 100K requests, with 512MB and 500ms average duration. Pricing per request is $0.2 per 1M requests, which would cost $0.02 cents for 100k requests [1]. Pricing for the computation in comparison is much higher, with 100K * 0.5 * (0.5/1)= 25,000 seconds of computation needed. Lambda's standard x86 price is around $0.00001667, meaning that the computational cost is approximately $0.42 a month [1]. The total cost needed for Lambda is $0.02 + $0.42 = $0.44 per month.

Given that Lambda, at a worst case, 100K requests, API Gateway would likely have 100K requests as well. Their pricing is $1.00 per million requests, and with 100K requests, this would equate to $0.10 per month [2].

DynamoDB's worst-case scenario would likely be 2x the requests of Lambda (200K) for reading, with 50K writes, and 1GB to store user data through tables [4]. DynamoDB's pricing for storage is $0.25 per GB, meaning a cost of $0.25. Writing costs $1.25 per million, and with 50K writes, this equates to $0.06 per month [4]. Reading costs $0.25 per million [4]. For 200K reads, the cost amounts to $0.05, meaning DynamoDB costs $0.25 + $0.06 + $0.05 = $0.36 per month.

Assuming that S3 at a worse case, needs 10GB storage, with 20K PUT and 100K GET requests, the cost breakdown is as follows. S3 costs $0.0023 per GB, meaning that for 10GB it would cost $0.23 to store all user documents [5]. PUT is $0.005 per 1000 requests, meaning a cost of $0.10, and GET is $0.0004 per 1000 requests, meaning a cost of $0.04 [5]. The total cost of S3 would be around $0.37.

Given Gemini API's free tier that allows for 1500 free requests per day, 45,000 free requests a month, this allows for a huge buffer when only estimating 20,000K requests per month [6]. This would cost $0.

In total, to keep DocAI Organizer ongoing with 1000 monthly users, it would cost $0.44 + $0.10 + $0.36 + $0.37 = $1.27 a month at the worst case. The actual cost would likely be much lower than this, meaning an extremely affordable ongoing price.

5.3 Additional Costs

There are a couple of other things to consider for public release, which has its own separate benefits and costs. CloudWatch Logs allows us to look at any logs made by Lambda to catch any errors or bugs, with real-time monitoring, audit and security. It allows us to oversee almost everything done, which could help with debugging errors or finding security gaps. The cost of CloudWatch is 0.50GB per GB, and with 1000 users monthly, these logs would cost us close to nothing [7]. Secondly, buying our own domain helps more users access our website more easily and portrays professionalism. Buying a .com domain would cost, at the

worst-case scenario, around $100 a year, and with the cost of Route 53's 25 hosted zones being $0.5 a month, this would cost us approximately $16.70 a month [8].

5.4 Cost Comparison

Since EC2 was the most heavily considered alternative service used, we will compare the cost of the current architecture of the serverless Lambda + DynamoDB with EC2 + RDS. Since the main operations of the application aren't too heavy on computational power, a safe and conservative EC2 instance would be t3.micro when running 24/7 with a 1-year reserved instance pricing. would cost around $10 a month alone [9]. Adding RDS would cost at worst case $20 per month for a db.t3.micros instance [10]. This approach would cost around $30 a month, which is significantly more expensive than the current architecture and likely more inefficient.

Given the nature of document organization being a task, Lambda fits the use case perfectly. The baseline cost while idle using Lambda is near-zero, and it scales perfectly well as usage grows. The serverless approach costs $1.27, whereas the EC2 + RDS approach costs 23 times more for the same functionality. The only way I see this approach becoming cheaper than the current structure is if website traffic becomes extremely high. For the current plan for DocAI Organizer, Lambda makes the most sense.

5.5 Cost Optimization

With the current state of DocAI Organizer, there are two big optimizations that could significantly reduce the cost of ongoing costs. Right now, there is no way for users to delete their accounts. This functionality could open space in S3 as well as DynamoDB, which could also reduce the cost of holding that user's data. Another big optimization that could play a bigger role in reducing cost is batch AI processing. Right now, Gemini handles a single file at a time when organizing, which is not as efficient nor cost-effective as doing multiple files at a time. This would significantly lower the number of API calls, which then leads to lower costs.

6. Evolution

Given the nature of this cloud application, and the technology stack of Lambda, S3, DynamoDB and API Gateway, a lot of this cloud application is complete. All four technologies have auto-scaling capabilities, so in the case that thousands of users start using DocAI Organizer every day, these technologies can scale up with almost zero complications.

6.1 Technical Debt

There is one important feature to add in the future. Currently, there is no way for a user to reset their password in the case that they forget it. This is an extremely common feature for almost every service online, so this is the biggest priority to add in the future. Unfortunately, due to technical debt, it could not be implemented by the due date, as the core flow and other user stories took a lot of refining. To implement a forgot password functionality, a new AWS service called Simple Email Service (SES) would be implemented. SES is commonly used in the industry to send transactional emails like receipts and marketing emails like newsletters, and even the exact functionality I'm looking to implement, a reset password email. SES works in harmony with the current stack as it can be directly used in lambda, is extremely cheap, and it follows the serverless architecture DocAI Organizer currently has, as SES doesn't have an email server to manage like other alternatives.

6.2 Future Proofing

With the advancements of AI seemingly being exponential with each passing year, it won't be long until a model that outperforms Google Gemini's 2.5 Flash with higher accuracy and cheaper rates is available. This new hypothetical model will replace the current model to ensure that this application stays up to date with technological advancements and doesn't become obsolete. This loop of replacing the underlying AI model until a better alternative is available will continue until the discontinuation of DocAI Organizer.

6.3 Final Remarks

I believe that with this application being a document organizing AI, it's important to stay true to its function. A lot of services like to become bigger than their original purpose, which leads to a degradation in the quality of the function they were first set out to perform. This is for organizing documents, and that alone. We want one working tool over multiple fewer effective tools. Staying true to the function of this website is important. The design philosophy of this cloud application is that less is better, and because of this, future features are sparse to ensure DocAI Organizer's primary function is as good as it can be.

Bibliography

[1] AWS, “AWS Lambda – Pricing,” *Amazon Web Services, Inc.*, 2024. https://aws.amazon.com/lambda/pricing/
[2] “Amazon API Gateway Pricing | API Management | Amazon Web Services,” *Amazon Web Services, Inc*. https://aws.amazon.com/api-gateway/pricing/
[3] Amazon, “AWS Free Tier,” *Amazon Web Services, Inc.*, 2019. https://aws.amazon.com/free/
[4] “Amazon DynamoDB Pricing for On-Demand Capacity,” *Amazon Web Services, Inc*. https://aws.amazon.com/dynamodb/pricing/on-demand/
[5] AWS, “Amazon S3 Pricing,” *Amazon Web Services, Inc.*, 2025. https://aws.amazon.com/s3/pricing/
[6] Google, “Gemini Developer API Pricing,” *Google AI for Developers*, 2025. https://ai.google.dev/gemini-api/docs/pricing
[7] “Amazon CloudWatch Pricing – Amazon Web Services (AWS),” *Amazon Web Services, Inc*. https://aws.amazon.com/cloudwatch/pricing/
[8] AWS, “Amazon Route 53 pricing - Amazon Web Services,” *Amazon Web Services, Inc*. https://aws.amazon.com/route53/pricing/