Serverless PDF Translation Platform on AWS

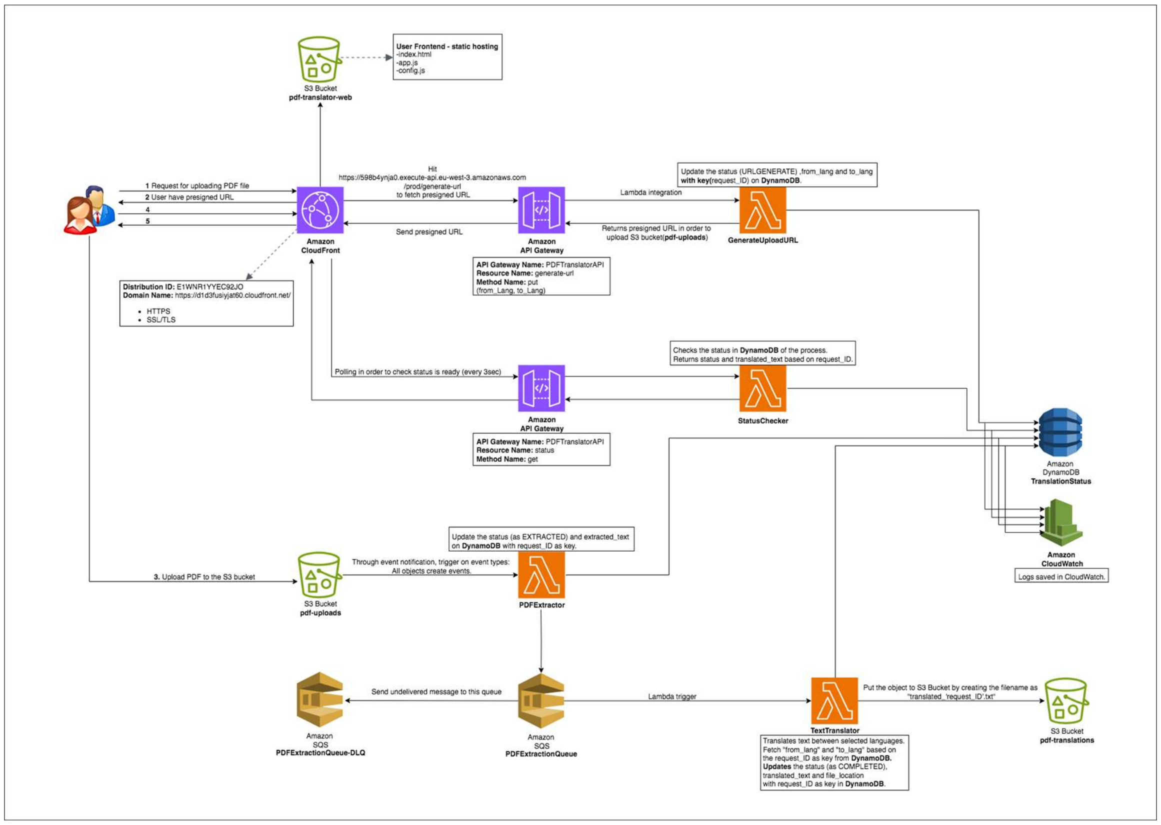
This project was carried out as a compulsory part of the Cloud Computing course under Professor Simone Merlini and Professor Nicolo Marchesi.

1. Introduction

This project focuses on the design and implementation of a serverless PDF translation platform using Amazon Web Services (AWS). The system enables users to upload PDF documents, extract text automatically, translate the content into a selected target language, and retrieve the translated output through a web-based interface.

The project evolved gradually from a fully manual deployment approach to a complete CI/CD-based deployment pipeline. While the deployment methodology improved significantly over time, the core backend architecture remained consistent. This report presents the final system architecture, workflow, and deployment evolution in a concise and structured manner.

2. System Architecture Overview


S3 Bucket
pdf-translator-web
User Frontend - static hosting
-index.html
-app.js
-config.js
1 Request for uploading PDF file
2 User have presigned URL
4
5
Amazon
CloudFront
Hit
https://598b4ynja0.execute-api.eu-west-3.amazonaws.com
/prod/generate-url
to fetch presigned URL
Send presigned URL
Amazon
API Gateway
Lambda integration
Returns presigned URL in order to
upload S3 bucket(pdf-uploads)
Update the status (URLGENERATE) ,from_lang and to_lang
with key(request_ID) on DynamoDB.
GenerateUploadURL
API Gateway Name: PDFTranslatorAPI
Resource Name: generate-url
Method Name: put
(from_Lang, to_Lang)
Distribution ID: E1WNR1YYEC92JO
Domain Name: https://d1d3fusiyjat60.cloudfront.net/
• HTTPS
• SSL/TLS
Polling in order to check status is ready (every 3sec)
Amazon
API Gateway
Checks the status in DynamoDB of the process.
Returns status and translated_text based on request_ID.
StatusChecker
API Gateway Name: PDFTranslatorAPI
Resource Name: status
Method Name: get
Amazon
DynamoDB
TranslationStatus
Amazon
CloudWatch
Logs saved in CloudWatch.
3. Upload PDF to the S3 bucket
S3 Bucket
pdf-uploads
Through event notification, trigger on event types:
All objects create events.
Update the status (as EXTRACTED) and extracted_text
on DynamoDB with request_ID as key.
PDFExtractor
Send undelivered message to this queue
Amazon
SQS
PDFExtractionQueue-DLQ
Amazon
SQS
PDFExtractionQueue
Lambda trigger
TextTranslator
Translates text between selected languages.
Fetch "from_lang" and "to_lang" based on
the request_ID as key from DynamoDB.
Updates the status (as COMPLETED),
translated_text and file_location
with request_ID as key in DynamoDB.
Put the object to S3 Bucket by creating the filename as
"translated_'request_ID'.txt"
S3 Bucket
pdf-translations

The frontend is implemented as a static website hosted on Amazon S3 and distributed globally using Amazon CloudFront with HTTPS support. It provides users with the ability to upload PDF files and monitor the translation status.

The backend is composed of multiple AWS-managed services. Amazon API Gateway exposes REST endpoints for generating pre-signed upload URLs and querying translation status. AWS Lambda functions handle all processing logic. Uploaded PDFs and translated outputs are stored in Amazon S3. Amazon DynamoDB maintains translation metadata and status. Amazon SQS decouples text extraction from translation, enabling asynchronous processing. Amazon CloudWatch is used for logging and monitoring.

3. End-to-End Workflow

The end-to-end workflow begins when the user accesses the frontend application through Amazon CloudFront. The static frontend is hosted on Amazon S3 and provides an interface for uploading PDF files and monitoring translation progress.

Step 1: Request for Upload URL

When the user initiates a PDF upload, the frontend sends a request to the Amazon API Gateway endpoint responsible for generating a pre-signed URL. API Gateway invokes the GenerateUploadURL Lambda function. This function creates a unique request ID, stores the initial status (URL_GENERATED) along with source and target language metadata in Amazon DynamoDB, and returns a pre-signed S3 URL to the user.

Step 2: PDF Upload to S3

Using the received pre-signed URL, the user uploads the PDF file directly to the S3 upload bucket. This direct upload avoids routing large files through backend services and improves scalability. Once the file is uploaded, an S3 Object Created event is triggered.

Step 3: PDF Text Extraction

The S3 event automatically invokes the PDFExtractor Lambda function. This function downloads the PDF, extracts the text content, and updates the DynamoDB entry with status EXTRACTED along with the extracted text. After successful extraction, a message containing the request ID is sent to an Amazon SQS queue to initiate the translation phase.

If the PDF extraction fails, the message is retried according to SQS retry policies. After exceeding the maximum retry attempts, the message is moved to the Dead Letter Queue (DLQ). The DLQ allows failed extraction requests to be isolated for debugging and prevents repeated failures from blocking the system.

Step 4: Text Translation

Messages in the SQS queue trigger the TextTranslator Lambda function. This function retrieves the extracted text and language parameters from DynamoDB and performs translation using Amazon Translate or an equivalent translation model. The translated output is stored as a text file in the S3 translation bucket. DynamoDB is then updated with status COMPLETED and the S3 file location.

If translation processing fails, the message is retried automatically. Failed translation messages are eventually redirected to the DLQ, ensuring reliability and fault isolation.

Step 5: Status Monitoring and Result Retrieval

While backend processing occurs asynchronously, the frontend periodically polls the status endpoint exposed by API Gateway. The StatusChecker Lambda function queries DynamoDB using the request ID and returns the current translation status and output location to the user.

Step 6: Logging and Monitoring

All Lambda executions, API Gateway requests, and system errors are logged in Amazon CloudWatch. CloudWatch logs enable real-time monitoring, troubleshooting, and performance analysis of the entire workflow, including failures routed to the DLQ.

This event-driven and asynchronous workflow ensures that long-running tasks do not block user requests while maintaining reliability, scalability, and fault tolerance.

4. Deployment Evolution

The project deployment evolved across four major stages.

Initially, all AWS resources were created manually through the AWS Management Console. Lambda functions were uploaded as ZIP files, and CloudFront cache invalidation was performed manually. Although useful for learning, this approach was time-consuming and error-prone.

In the second stage, partial automation was introduced using the AWS CLI for CloudFront invalidations. However, backend resources were still manually managed.

The third stage marked the introduction of Infrastructure as Code using AWS CloudFormation. Automation scripts were developed to package Lambda functions, upload artifacts, and deploy the entire backend stack consistently. This significantly improved reproducibility and reliability but still relied on local execution.

The final stage implemented a full CI/CD pipeline using GitHub Actions. All deployment steps, including dependency installation, Lambda packaging, artifact uploads, and CloudFormation deployment, were automated in the cloud. AWS credentials were securely managed using GitHub Secrets, resulting in a production-ready deployment process.

5. Key Design Benefits

The final system offers several advantages. The serverless architecture eliminates server management and scales automatically. Asynchronous processing allows efficient handling of large PDF files. Loose coupling through SQS improves fault tolerance. The pay-per-use pricing model ensures cost efficiency. The use of Infrastructure as Code and CI/CD reflects modern DevOps best practices.

6. Conclusion

This project demonstrates a complete transformation from a manually deployed prototype to a scalable, automated, and production-ready serverless application. By leveraging AWS managed services and modern DevOps practices, the platform achieves high reliability, maintainability, and efficiency. The final solution provides a strong foundation for future enhancements such as authentication, multilingual expansion, and advanced monitoring.