CSCI 5411 – Final Project

Milestone-1

Smart Expense Tracker

Name: Shray Moza

ID: B00987463

1. Application Domain Selection & Justification

Domain: Finance – Cloud-Based Smart Expense Tracking System
I selected this specific domain because manual receipt tracking is a real-world problem I faced, and it presents a perfect opportunity to implement an event-driven architecture. Most users struggle with organizing receipts, extracting data manually, and monitoring monthly spending. Traditional mobile apps require manual data entry and often lack automation or reliable OCR capabilities.

This allows exploration of many AWS services like S3, Lambda Textract, SNS, Cognito, etc. while remaining cost-effective for a personal account. It requires high availability and on-demand scaling, which serverless services provide without the fixed costs of running dedicated EC2 instances. This project allows me to use Amazon Textract for Text Extraction and processing.

---

2. Problem Statement & Business Case

Users often keep physical receipts or email invoices, but manually adding expenses is slow, error-prone, and time-consuming. Tracking expenses across months and maintaining categories manually leads to financial oversight and poor budget control.

My solution focuses on three things:

- Auto-extraction of expense details from uploaded receipts using AWS Textract.
- Auto-categorization of expenses using a rule-based engine.
- Monthly expense aggregation for threshold monitoring so that users are notified when spending crosses a set limit.
- A simple interface deployed via AWS Amplify, making the system globally accessible.
- Serverless architecture for low cost and easy deployment.

---

3. Functional Requirements

1. **User Authentication**: Users can sign up, log in, and manage their account using Amazon Cognito.
2. **Receipt Upload**: Users can upload receipts images or PDFs from the Amplify-hosted web app. The frontend requests a presigned S3 upload URL through API Gateway and Lambda.
3. **Automated OCR Processing**: When a receipt is uploaded, a Lambda function is triggered. Lambda uses Textract to extract key information like Total amount, Date, Store/Merchant name, Item details if available.
4. **Automatic Categorization**: Expenses are categorized using rules like Walmart in Groceries, Uber in Transport etc. Users can update categories if the automatic classification is incorrect.

5. **Expenses Dashboard**: Users can view all their expenses, filter by category, and see metadata.
6. **Monthly Summary**: Monthly total spending is automatically aggregated.
7. **Threshold Alerts**: If monthly spending crosses the user-set threshold SNS sends email to user.
8. **Update Category**: Users can edit incorrect categories. Admin can update rule table for auto-categorization.

---

4. Non-Functional Requirements

4.1 Availability

Completely Serverless architecture using Lambda, S3, DynamoDB etc. provides high availability without administrative overhead.

4.2 Scalability

Components scale automatically:

- **S3:** Capable of storing thousands of receipt images without pre-provisioning storage.
- **Lambda:** Can process concurrent receipt uploads simultaneously without queuing.
- **DynamoDB:** On-Demand mode handles the 'bursty' traffic nature of expense reporting.

4.3 Performance

Textract processes most receipts in a couple of seconds, which keeps the application responsive. Using API Gateway ensures low latency API responses.

4.4 Reliability

Event-driven S3 to Lambda workflow automatically retries failures. Also, DynamoDB offers multi-availability zone durability.

4.5 Operations / Monitoring

- CloudWatch logs for all Lambdas.
- CloudWatch metrics for monthly spending.
- CloudWatch alarms for multi-user cost thresholds.

4.6 Cost Optimization

Serverless model keeps costs low as:

- DynamoDB On-Demand
- Lambda pay-per-use
- Textract billed per page
- Amplify hosting inexpensive for small projects

Estimated Monthly Cost Breakdown (Development Phase)

Service	Pricing Model	Estimated Usage (Dev/Demo)	Monthly Cost Estimate
AWS Lambda	Pay-per-request ($0.20/1M requests)	500 invocations (Approx)	$0.00 (Free Tier covers 1M/mo)
Amazon S3	Storage ($0.023/GB)	< 100 MB (Receipt images)	$0.00 (Free Tier covers 5GB)
DynamoDB	On-Demand ($1.25/1M writes)	1,000 Read/Write units (Approx)	$0.00 (Free Tier covers 25GB storage)
Amazon Textract	Per page processed ($1.50/1k pages)	50 receipts (Approx)	$0.00 (Free Tier covers 1k pages/mo)
API Gateway	Per million calls ($1.00/1M calls)	500 API calls (Approx)	$0.00 (Free Tier covers 1M calls)
Amazon Cognito	Monthly Active Users (MAU)	1-5 Users	$0.00 (Free Tier covers 50k users)
CloudWatch	Logs ($0.50/GB) & Alarms	< 100 MB logs	$0.00 (Free Tier covers 5GB & 10 alarms)
Amplify Hosting	Data Transfer ($0.15/GB)	< 1 GB transfer	$0.00 (Free Tier covers 15GB/mo)
Total			Around $0.00 - $0.10

4.7 Security

Since I will be using my personal AWS account, I will have full control over security configurations:

- **Authentication:** Amazon Cognito will handle all user sign-ups and token generation.
- **Least Privilege:** The Lambda functions will have strict IAM roles. For example, the ProcessReceipt Lambda will be allowed Put Item access *only* to the specific ExpensesTable and not the whole database.
- **Data Privacy:** S3 buckets will be public access blocked, with uploads only allowed via short-lived presigned URLs generated by the backend.

4.8 Disaster Recovery

While AWS handles physical availability, I remain responsible for data protection:

- **S3 Versioning:** I will enable versioning on the Receipts Bucket. If a user or a script accidentally deletes or overwrites a receipt image, the original version can still be recovered.
- **Infrastructure as Code:** By using Terraform, my entire infrastructure can be redeployed in a different region within minutes if the existing region suffers a total outage.

---

5. AWS Well-Architected Framework Alignment

This section explains how my design choices align with the six pillars of the AWS Well-Architected Framework:

1. Operational Excellence

The system is designed to be observable. Instead of just logging, I plan to configure CloudWatch Alarms to trigger if the ProcessReceipt Lambda errors out more than 5 times in a minute. This alerts me immediately if a bad code update breaks the OCR pipeline. I am also using Infrastructure as Code (Terraform) to ensure I can tear down and rebuild the environment without manual console clicks, reducing human error.

2. Security

Security is a priority, especially since this account is tied to my personal billing.

- **Identity Management:** Unlike the restricted Learner Lab environment, I will implement granular IAM policies. Public access to S3 buckets will be explicitly blocked.
- **Data Boundary:** By using Cognito for authentication, I ensure that user data is isolated. The DynamoDB table design uses the UserID as the partition key, ensuring users can only query their own expenses.

3. Reliability

I am using an event-driven design to decouple components. If the OCR process Textract is slow, it does not block the user's upload experience because the processing happens asynchronously. I also plan to implement a Dead Letter Queue for the Lambda. If a receipt fails to process after 3 retries, the event is sent to the DLQ so I can investigate it later without losing the data.

4. Performance Efficiency

I have selected DynamoDB On-Demand mode. Since I cannot predict the exact traffic volume during the grading period, On-Demand ensures the database automatically accommodates spikes in requests without me having to manually guess read/write capacity units. This aligns performance directly with usage.

5. Cost Optimization

Since I will be using my personal account, avoiding fixed costs will be a strict requirement.

- **Zero-Scale Architecture:** I chose a purely serverless stack like Lambda, S3, DynamoDB On-Demand. If no receipts are uploaded, my bill is effectively less than $10.00 per week.
- **Network Cost Avoidance:** I have deliberately excluded VPC NAT Gateways and Interface Endpoints. While they offer network isolation, they incur a fixed hourly cost approx. $30/month that is unnecessary for a student project with low traffic.
- **Budget Alerts:** I will configure AWS Budgets to send me an email if my forecasted monthly spend exceeds $15.00, preventing accidental cost overruns.

6. Sustainability

My serverless design maximizes energy efficiency. Because I am not reserving capacity like provisioned DynamoDB or running EC2 instances, AWS does not need to keep hardware powered on for my application when no one is using it. Resources are only provisioned for the few milliseconds it takes to process a receipt, which is the most energy-efficient way to run this workload.

---

6. Initial Architecture Design (Made in Draw.io)

6.1 Architecture Diagram

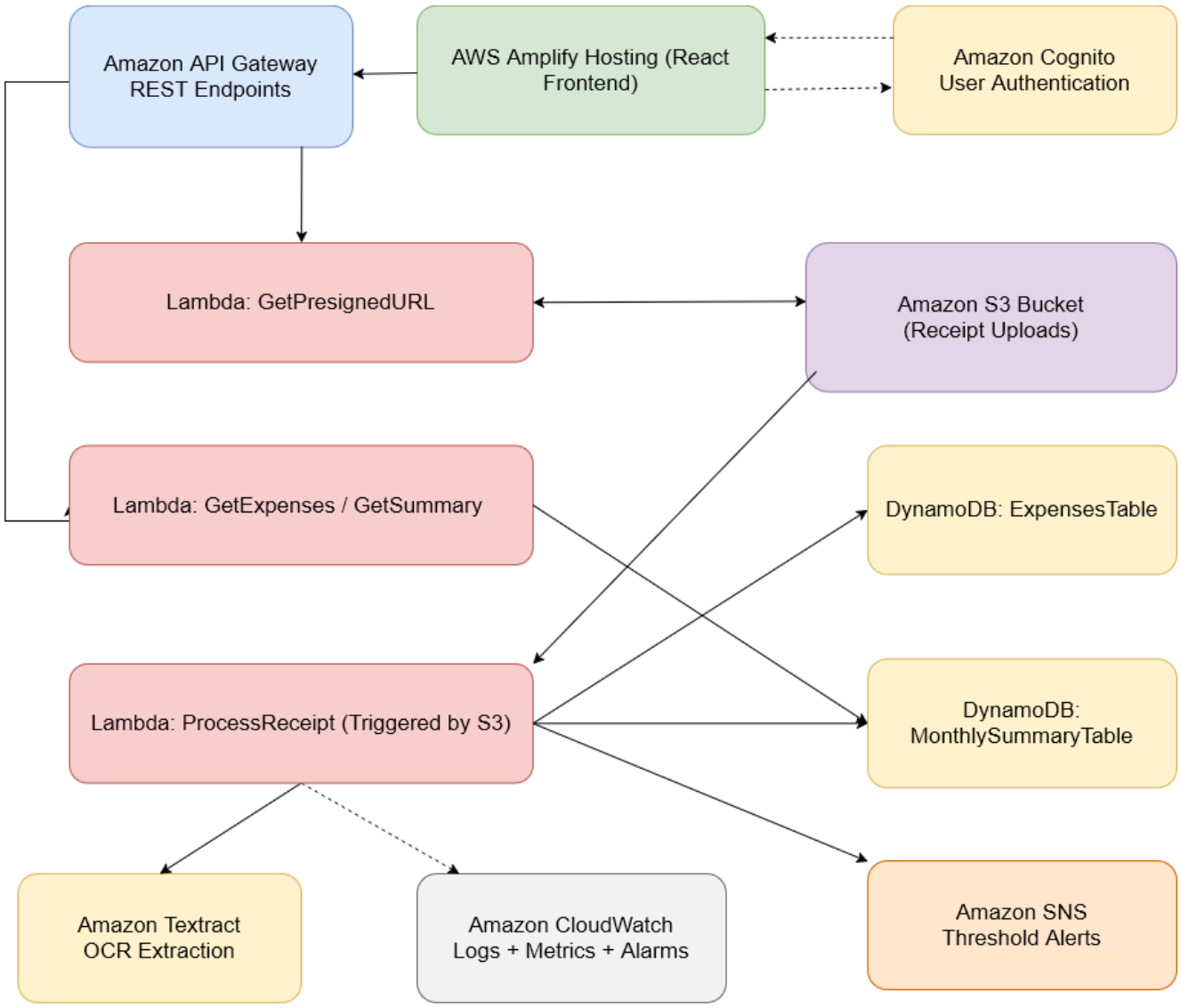

Figure 1: Architecture Diagram

6.2 Architecture Components

1. Frontend: Plan to Build using React and host on AWS Amplify Hosting. Integrated with Amazon Cognito for auth.

2. Authentication Layer: Amazon Cognito User Pool: For sign-in, sign-up, authentication tokens.

3. Storage Layer: Amazon S3 (ReceiptsBucket): Stores uploaded receipts and receives files using presigned URLs.

4. Backend Compute Layer

- Lambda GetPresignedURLFunction: Returns upload URL for S3.
- Lambda ProcessReceiptFunction: Triggered on S3 upload, calls Textract, extracts data, categorizes data, writes to DynamoDB, triggers SNS if threshold crossed.
- Lambda GetExpensesFunction: Returns list of user expenses.
- Lambda GetMonthlySummaryFunction: Returns aggregate totals.

5. OCR Layer: Amazon Textract: Extracts structured data from receipts.

6. Database Layer

- DynamoDB (ExpensesTable): Stores each extracted expense.
- DynamoDB (MonthlySummaryTable): Stores monthly spendings, threshold and notification flags.
- DynamoDB (CategoryRulesTable): Stores custom rules for categorization.

7. Monitoring / Notification Layer

- Amazon SNS: Sends threshold alert emails/SMS.
- Amazon CloudWatch: Logs, metrics, and alarms.

8. Networking & Integration

API Gateway's:

- /presign-upload
- /expenses
- /summary
- /update-category

---

7. Data Sequence Diagram (Made in Draw.io)

7.1 Sequence Diagram

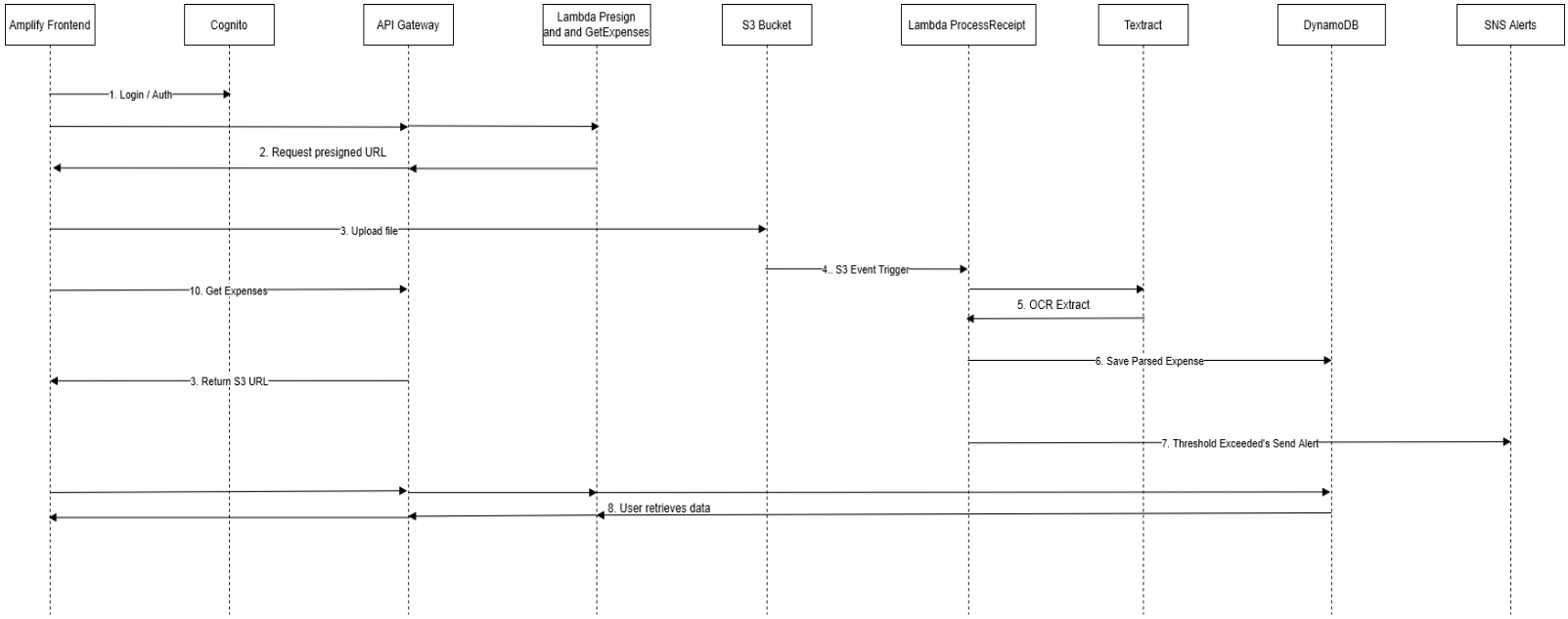

Figure 2: Sequence Diagram

7.2 Sequence Flow

1. Login: Amplify requests tokens from Cognito.

2. Request presigned URL: Amplify connects through API Gateway to Lambda to request presigned URL.

3. Upload file: Amplify uploads to S3 using presigned URL.

4. S3 triggers backend: S3 triggers Lambda ProcessReceipt.

5. OCR extraction: Lambda ProcessReceipt uses Textract to extract text and then processes the response.

6. Save parsed results: Lambda ProcessReceipt saves outputs in DynamoDB (ExpensesTable & MonthlySummaryTable).

7. Threshold notification: Lambda ProcessReceipt triggers SNS Alerts in case of threshold breach.

8. User retrieves data: Amplify connects using API Gateway to GetExpenses Lambda that fetches data from DynamoDB and returns outputs through the same path.

---

8. AWS Services & Tech Stack

8.1 AWS Services Used

Category	Service	Purpose
Compute	AWS Lambda	Execution of presigning logic, OCR processing, and threshold evaluation.
Storage	S3	Receipt storage
Database	DynamoDB	Expenses, rules, summaries
ML	Textract	OCR extraction
Networking & Delivery	API Gateway	Expose APIs to frontend
Application Integration	S3 Trigger, SNS	Event processing & notifications
Security	Cognito, IAM	Authentication & permissions
Monitoring	CloudWatch	Logs & Alarms
Developer Tools	AWS Amplify	Frontend hosting & CI/CD

8.2 Tech Stack

1. **Frontend**: React with Amplify hosting
2. **Middleware**: API Gateway
3. **Backend**: Lambda
4. **Database**: DynamoDB
5. **Infrastructure**: Terraform
6. **Auth**: Cognito
7. **OCR**: Textract

---

9. Potential Architectural Challenges

1. **Textract edge cases**: Some receipts may be handwritten/ blurry hence extraction quality varies.
2. **Categorization accuracy**: Rule-based engine requires admin updates to improve classification.
3. **Ensuring Idempotency**: S3 to Lambda events may retry so Lambda must avoid duplicate writes.
4. **DynamoDB table design**: PK/SK must be well-designed for efficient queries.
5. **Textract Costs:** Textract is billed per page. To prevent accidental high bills, I need to ensure the Lambda only processes valid image formats and implement a limit on how many receipts a user can upload per day.
6. **API Throttling:** Since I am paying for usage, I must configure Throttling on the API Gateway (e.g., 10 requests per second) to prevent a malicious user from flooding my API and driving up my Lambda execution costs.
7. **Asynchronous User Experience:** Since the OCR happens in the background, the frontend needs to handle the delay gracefully.
