Project Proposal

EE 547: Applied and Cloud Computing for Electrical Engineers

Project Title: ReceiptInbox — Serverless Receipt Parser, Spend Categorizer, and Smart Alerts

Team Members: Harry Kang, Lin Lin Hua, Vivin Thiyagarajan

Introduction

People collect receipts in different formats — screenshots, photos, email attachments, and PDF invoices — but they only need a simple view: *what did I spend, on what, and when?* Manually transcribing this information into a spreadsheet is error-prone and usually gets skipped.

ReceiptInbox is a cloud-hosted web application that automates receipt management. A user uploads (or later, forwards) a receipt. The system extracts merchant, date, subtotal, tax, and total, tries to infer the spending category, and stores everything in a searchable dashboard. On top of that, it runs simple anomaly rules (for example: “this looks like a duplicate” or “this total is unusually high for this user”) and notifies the user.

Functionality:

1. Upload receipts from browser
2. Automatic ORC and parsing
3. Category suggestion (e.g. Groceries, Travel)
4. Detect anomalies and push notifications (email/in-UI)
5. Searchable receipts with details

Potential users: Individuals or small businesses looking to automate expense tracking, reduce manual data entry, and gain insights from their spending patterns.

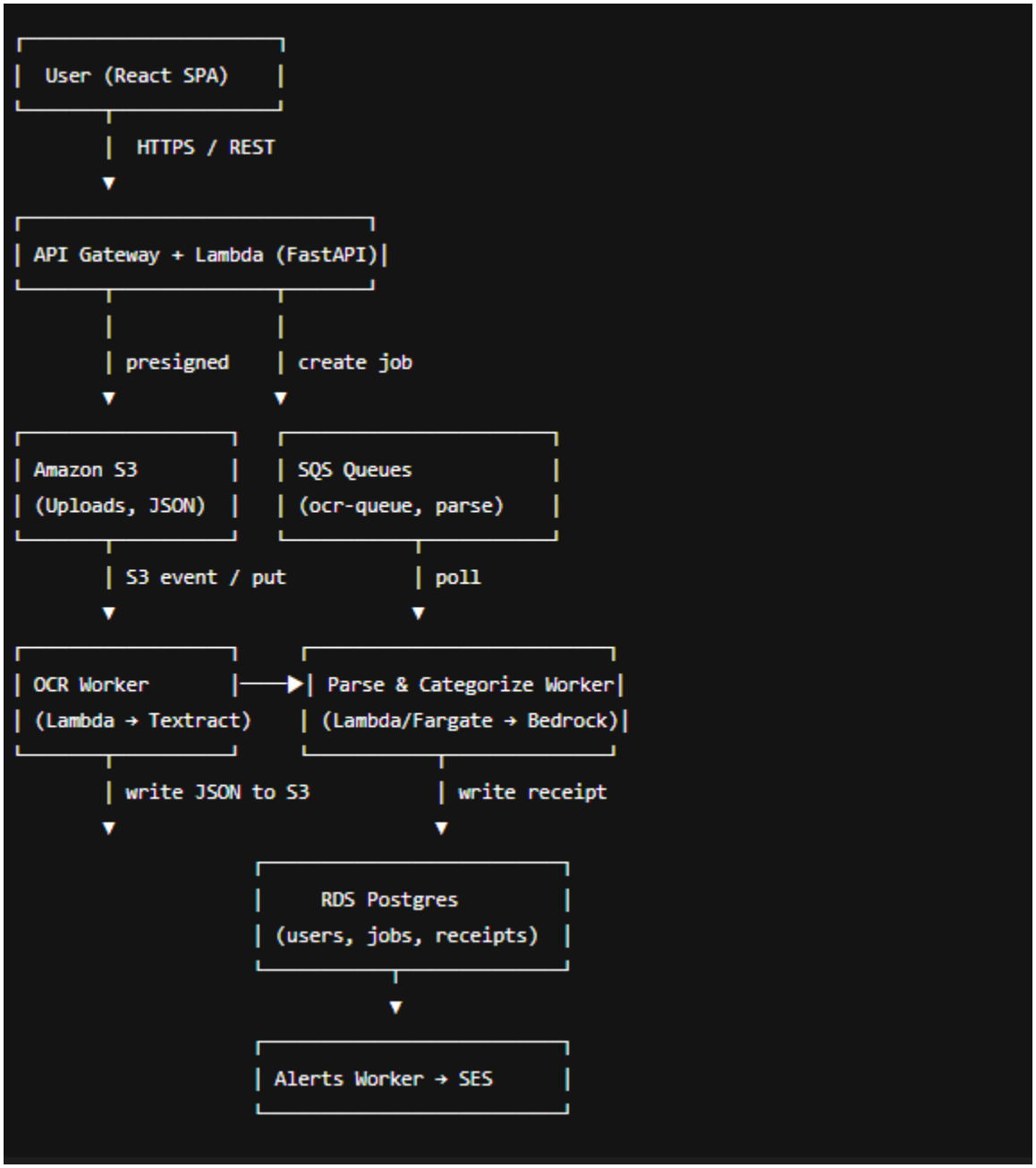
Why cloud computing: This problem is appropriate for a cloud computing project because receipt processing is inherently bursty and heterogeneous (involving OCR, parsing, embedding, rules engines). This makes it suitable for asynchronous pipelines using queue-based fan-out and serverless computing. It requires multiple integrated components (API, ML services, database, notification service) and asynchronous processing (e.g., parsing a PDF can be slow), all of which are core to AWS services.

System Architecture

Component Overview

- **User Interface (React)**: A web application allowing users to signup/login, upload receipt, review and edit processing results (e.g. scanned text, suggested categories).
- **User requests and API endpoints**: Amazon API Gateway with AWS Lambda (API Handler). Handles user authentication and receipt upload requests.
- **Asynchronous background processing**: Amazon SQS (for queue-based fan-out) and AWS Lambda (Workers). An SQS queue receives upload tasks, which are processed by Lambda workers.
- **Data storage**: Amazon S3 for raw receipt files (PDF/JPG). Amazon DynamoDB and PostgreSQL on EC2 for user information, extracted structured data, and categorization results.
- **ML functionality**: We will use AWS-managed AI services:
    - AWS Textract: For OCR and extraction of line items and totals.
    - Amazon Comprehend: For NLP to categorize items ("Groceries," "Travel").
- **External service integration (Alerts)**: Amazon SNS (Simple Notification Service) to send anomaly notifications to users.

Architecture Diagram



Technology Stack

Languages:

- **Python** for all backend AWS Lambda functions.
- **JavaScript (React)** for the frontend UI.

Databases and Storage:

- **Amazon S3** for raw file storage.
- **Amazon EC2 (PostgreSQL)** for joining across user data, receipts, line items, and alerts.
- **Amazon DynamoDB** for extracted receipt metadata, as it is ideal for key-value access in a serverless architecture.

AWS Services:

- API Gateway, AWS Lambda, Amazon SQS, Amazon S3, Amazon EC2**,** DynamoDB, AWS Textract, Amazon Comprehend, Amazon SNS, AWS IAM (for permissions).

Frameworks:

- FastAPI, React, SQLAlchemy, Alembic, Pydantic, TailwindCSS, TanStack Query

Communication:

- **REST APIs** – client-server communication
- **S3 events** – kick off OCR
- **SQS** – queues for asynchronous communication between the API handler and backend workers
- **Direct DB queries** – for UI reads

Technical Details

REST API

Endpoints:

- POST /auth/signup → create user, return JWT
- POST /auth/login → validate, return JWT
- POST /receipts/upload-url → returns { url, s3_key } to upload directly to S3
- POST /receipts → allows authenticated users to upload a receipt file and returns a task_id immediately.
- POST /alerts/rules → create/update alert thresholds

- GET /receipts → list receipts for the user
- GET /receipts/{id} → processing status, parsed receipt including line items and anomalies
- GET /dashboard → fetches aggregated and categorized spending data for dashboard visualization.

Authentication: Users can sign up through the web app, then the authentication information will be encrypted and stored in the database (PostgreSQL on EC2). When logging in, users' information will be sent to EC2 for authentication.

Error Handling: We will handle errors and return appropriate status codes (e.g., 400, 401, 500).

Asynchronous Processing

- **Why:** Operations like OCR (Textract) and ML inference (Comprehend) are slow and "bursty". Running them synchronously would lead to API timeouts. Failed jobs can be retired automatically without requiring users to be online the whole time.
- **Services:** Users can use API endpoints (i.e. POST /receipts) to upload receipt files to S3 and put a task onto an SQS queue. One or more Lambda workers will then perform designed tasks such as pulling messages, textracting, writing output to S3, sending notification to user, etc.
- **Status & Failures:** Users can interact with APIs (i.e. GET /receipts/{id}) to check status which will be stored in the database. Failures will be handled by SQS's built-in retry mechanisms and Dead Letter Queue (DLQ).

Machine Learning Component

- **Technique:**
    - OCR/text extraction using AWS Textract.
    - Text/category Classification using Amazon Comprehend.
    - Anomaly Detection to identify duplicate charges or price changes.
- **Model**: We will use AWS-managed pre-trained models (Textract, Comprehend) to "minimize model ops."
- **Inference**: Inference will happen asynchronously in the Lambda workers (parse & categorize).
- **Purpose**: ML serves the application's purpose by automating the tedious tasks of data extraction and categorization.

- **Workflow Overview**
    1. Textract extracts text contents from uploaded documents (receipts).
    2. Remove redundant information (e.g. ads, tax, store name).
    3. Split into words; remove stopwords(e.g. of, the, a); create bag-of-words and true labels (categories).
    4. Train a multi-class classification model using Amazon Comprehend.

Data Persistence

- **Data Stored:** User account info (PostgreSQL), raw receipt files (S3), extracted structured data (items, totals), categories, and detected subscriptions (DynamoDB). Jobs info (status).
- **Systems:** S3 for file storage, DynamoDB and PostgreSQL for structured metadata.
- **Data Model:**
    - User (id, email, created_at)
    - Job (id, user_id, s3_key, status, created_at)
    - Receipt (id, user_id, data, subtotal, is_subscription)
    - Receipt_item (id, receipt_id, description, qty, price, category)
    - Alert_rules (id, user_id, type, is_active)
    - Alert_event (id, user_id, rule_id, receipt_id, message)
- **Access:** Lambda functions will use the AWS SDK and IAM roles to access DynamoDB and S3.

User Interface

- **Technology:** Frontend uses React Single Page Application (SPA).
- **Functionality:** User login/signup (PostgreSQL). Receipt upload interface. Dashboard for visualizing spend categories. List and detail views for processed receipts.
- **Authentication & Async:** Users can sign up/log in through web application UI (React); the authentication information will be encrypted and stored on the database.

Data Sources and External Services

- **Data Sources:** Data is generated by users uploading their own receipts.

- External dependencies include **AWS Textract (for OCR)**, and **SES (for email)**. If any service is unavailable, SQS retries or dead-letter queues ensure no data loss.

AWS Deployment

- **Compute: AWS Lambda** for all computing jobs (API handlers and async workers).
- **Storage: S3** buckets, **EC2 PostgreSQL**, **DynamoDB** tables (using on-demand capacity).
- **Networking:** API Gateway for public HTTP endpoints. Private VPC for database and workers. A small server running on EC2 for easier interaction with PostgreSQL.
- **Secrets: AWS Secrets Manager** or Lambda environment variables for credentials.
- **Deployment: AWS SAM** or **Serverless Framework** for infrastructure-as-code (IaC) deployment.

Implementation Plan

Team Responsibilities (subject to adjustment)

Harry Kang: Backend API (API Gateway + Lambda handlers), User Auth (PostgreSQL), S3/DynamoDB data model design.

Lin Lin Hua: Asynchronous processing (SQS + Lambda workers), ML integration (Textract, Comprehend), anomaly detection logic.

Vivin Thiyagarajan: Frontend UI (React), user dashboard, receipt upload interface, and API integration.

Milestones

By Revised Proposal (Nov 9): Set up S3, EC2, and SQS.

By Status Report (Nov 30): Deploy API Lambda with auth and upload endpoints; implement basic UI; add OCR Lambda using Textract; create Parse worker to classify and store receipts; connect frontend to backend.

By Demo (Dec 11): Add Alerts worker with SES; visualize anomalies and category summaries in dashboard; perform end-to-end testing.

Minimum Viable System

The minimum viable version of ReceiptInbox will implement a simplified end-to-end workflow that demonstrates the complete data flow with fewer moving parts. A single AWS Lambda function will handle both optical character recognition (OCR) and text parsing within the same execution, removing the need for intermediate message queues. The user interface will rely solely on periodic polling to check job completion rather than event-driven updates. Expense categorization will use simple pattern matching with regular expressions rather than embeddings or machine learning models. Only one alert rule—such as detecting unusually high totals—will be enabled. Despite these simplifications, the core architecture will remain intact, utilizing Amazon S3 for storage, PostgreSQL on Amazon EC2 for structured data, and the REST API layer for all user interactions. This ensures that even the minimal version preserves the same system design principles and deployment structure as the full application.

Technical Challenges

- **Receipt Quality/OCR Variability:** Some receipts may be blurry, wrinkled, or have poor lighting, which makes it hard for Textract to read the text correctly. We'll need to handle low-confidence outputs and allow easy re-checking of such cases.
- **Workflow Reliability:** Since the system uses multiple queues and background steps, one failed task could affect the whole process. Setting up retries and safe message handling will be important.
- **Scalability and Cost:** The app should handle many uploads without becoming expensive to run. Managing how often Lambdas run and how the database scales will help keep costs low. Idle costs are minimized by implementing serverless design.
  Data Privacy: Receipts can contain sensitive personal and financial details, so we need to make sure all files are private, encrypted, and only accessible to the right users.
- **Classification Accuracy:** Accurately mapping extracted merchant text to categories ("Groceries") is difficult and will require a mix of Comprehend and customized rules. Also detecting "duplicates" requires careful state management and logic.

Expected Outcomes

At project completion, ReceiptInbox will provide a fully functional cloud pipeline that automatically extracts and categorizes receipt data and notifies users of unusual spending patterns.

The application will demonstrate robust asynchronous workflows, real-time user interaction through API polling, and reliable ML-driven text extraction and classification.

Users will be able to search receipts, view spending by category, and receive email alerts for specific conditions. The architecture is extensible to handle email receipt ingestion or integration with financial dashboards in future work.

The project is technically interesting due to its scalable "queue fan-out" architecture and the integration of multiple AWS-managed AI services to solve a complex, real-world problem.