Final Report: Cloud-Native Scalable Personal Expense Tracker

Abhishek Balasaheb Bhingle
Indiana University Bloomington
abhingl@iu.edu

Shubham Sandip Salunke
Indiana University Bloomington
ssalunke@iu.edu

Aditya Nitin Pise
Indiana University Bloomington
anpise@iu.edu

***Abstract*—Accurate personal finance management requires timely capture and categorization of every expense, yet manual receipt entry is tedious, error-prone, and often delayed. This leads to incomplete records, misclassification, and reactive budgeting. We present a cloud-native expense tracking platform that automates receipt ingestion, text extraction, and visualization. A Python Flask backend deployed on AWS EKS invokes AWS Textract for OCR and o4-mini as the LLM parser to convert uploads into structured records. Raw receipt files are stored in Amazon S3, and parsed data is persisted in Amazon RDS, while built-in autoscaling ensures responsiveness under variable load. A React dashboard delivers real-time financial insights without manual intervention. Our end-to-end solution streamlines personal budgeting, reduces user effort, and empowers proactive expense awareness.**

I. Cloud Services

Service	Usage
AWS EKS	Container orchestration
Amazon S3	Object storage
Amazon RDS	Managed database
AWS API Gateway	API management
AWS IAM	Access control
Amazon CloudWatch	Monitoring & logging
Amazon Textract	OCR extraction

TABLE I
CLOUD SERVICES AND USAGE

II. Project Highlights

Sr. No.	Highlight
1	**End-to-End Automation:** React SPA + Flask backend; AWS Textract & o4-mini parsing; S3 & RDS storage; real-time visuals.
2	**Elastic Autoscaling:** Kubernetes HPA on AWS EKS scales 2–6 pods at 70% CPU/memory thresholds.
3	**Performance Evaluation:** Apache JMeter load tests (10–500 users) show 250ms median dashboard latency.
4	**Monitoring & Security:** AWS CloudWatch dashboards for metrics; API Gateway; IAM roles.

TABLE II
PROJECT HIGHLIGHTS

III. Introduction

Personal finance management remains tedious when reliant on manual receipt entry. Users typically photograph or scan receipts and then manually type vendor, date, and amount into budgeting tools—often long after the transaction—leading to missed or misclassified expenditures and reactive budgeting.

In this work, we introduce a cloud-native expense tracking platform that:

- Enables PDF or image receipt uploads through a React dashboard.
- Stores raw receipt files in Amazon S3 and runs a Python Flask backend deployed on AWS EKS.
- Invokes AWS Textract for OCR and o4-mini as the LLM parser to extract vendor, date, and total amount.
- Persists structured expense records in Amazon RDS.
- Provides real-time financial insights via dynamic charts in the React dashboard, eliminating manual data entry.

Our key contributions include:

- Development of a fully automated, cloud-native pipeline from ingestion to visualization.
- Deployment of a Python Flask application on AWS EKS with built-in Horizontal Pod Autoscaling.
- Seamless integration of managed AWS services (S3, RDS, API Gateway) to minimize operational overhead.

By leveraging managed Kubernetes on AWS EKS, AWS API Gateway for secure routing, and built-in Horizontal Pod Autoscaling (HPA), our platform achieves elasticity, fault tolerance, and enterprise-grade security without manual infrastructure provisioning. This end-to-end solution streamlines personal budgeting, reduces user effort, and empowers proactive expense awareness.

IV. Problem Overview

Current personal expense tracking systems require manual data entry, which is time-consuming and error-prone. Most tools lack automation and scalability. Traditional systems rely heavily on users to input data from receipts manually, which is not only time-consuming but also prone to human error. Additionally, these systems often lack the scalability and intelligence needed to handle large volumes of diverse receipt formats, leading to inconsistent records and limited financial insights.

Our solution addresses this by automating the entire expense processing pipeline using a cloud-native architecture. Users upload receipt files through a secure API Gateway, which are then stored in S3. A Kubernetes-hosted backend processes these files, using a combination of OCR and LLM models to accurately extract key data points such as vendor, amount, and date. The structured data is stored in a relational database (RDS), and users can access real-time visual insights through a responsive dashboard. This design minimizes manual effort, improves accuracy, and scales seamlessly with user demand.

V. Related Work

Manual expense tracking has long been the norm, where users store receipts physically or enter details into spreadsheets or budgeting apps. While this method offers control, it is prone to delays, misclassification, and loss of data—especially under frequent, small-value purchases. Recent solutions aim to digitize this process but still present several limitations.

Commercial platforms such as Expensify and Mint offer receipt scanning and bank integration. However, they rely on rigid templates or structured banking feeds. This limits their effectiveness when dealing with unstructured data (e.g., photos of receipts), and they lack semantic understanding of receipt content. Moreover, they do not offer detailed customization, and often require users to manually adjust errors in parsing or category mismatches.

Accounting tools like QuickBooks and Wave provide rule-based categorization and basic mobile OCR capabilities. However, these are optimized for small businesses, not individuals, and do not support autoscaling or flexible deployment. Furthermore, their OCR components are often proprietary black boxes, preventing custom enhancements or confidence-based routing.

Academic research has made strides in pairing deep learning OCR (e.g., CRNN, CRAFT) with transformer-based language models (e.g., BERT, LayoutLM) to improve layout understanding and semantic labeling. Notable works focus on invoice or receipt parsing under datasets such as SROIE or RVL-CDIP. However, such systems remain prototypes—often trained on limited data, not deployed at scale, and lacking integration with cloud-native infrastructure.

VI. Design

Our solution leverages a modular, containerized architecture deployed on AWS EKS to ensure scalability, resilience, and fault isolation.

1) **Frontend**
Built with React.js, the dashboard handles user registration/login, receipt uploads, and displays interactive Chart.js visualizations. Axios is used for authenticated API requests.

2) **Backend & API Gateway**
A Python Flask application provides a JWT-authenticated REST API for file uploads to Amazon S3, invokes AWS Textract and o4-mini for parsing, and is exposed securely via AWS API Gateway with HTTPS and CORS policies.

3) **OCR & Parsing Pipeline**
Receipts stored in S3 are processed by the Flask backend, which calls AWS Textract for OCR and then uses o4-mini to extract structured fields (vendor, date, total amount).

4) **Database & Infrastructure**
Parsed expense records are persisted in Amazon RDS (PostgreSQL). All components run on AWS EKS via Kubernetes manifests, with a Horizontal Pod Autoscaler that adjusts backend pod replicas based on CPU and memory utilization.

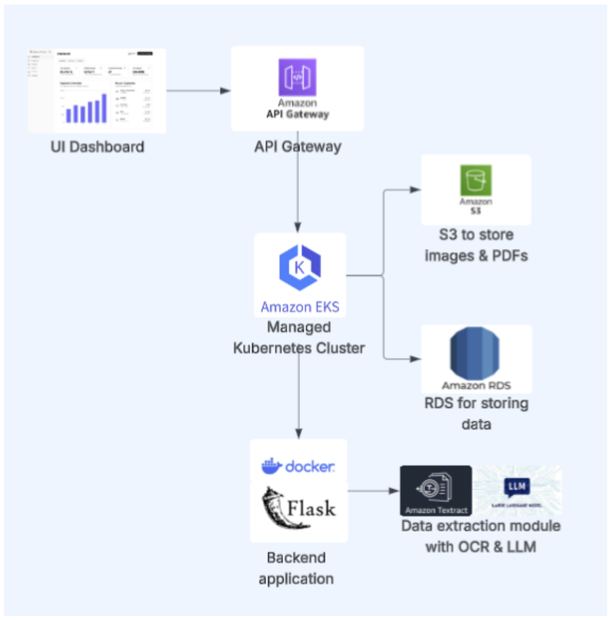

Fig. 1. System Architecture Diagram

VII. Implementation

We implemented the platform through the following stages:

1) **Local Backend Prototype**
Developed the Flask application against a local SQLite database. Integrated AWS Textract and o4-mini to verify end-to-end receipt parsing accuracy.

2) **AWS Service Integration**
Migrated storage to Amazon RDS for structured expense data and Amazon S3 for raw receipt files. Updated the Flask backend to use pre-signed S3 URLs and RDS connection parameters.

3) **Containerization and Local Testing**
Wrote Dockerfiles for the Flask backend and extraction module. Created Kubernetes manifests (Deployments, Services, HPA). Deployed to Minikube and validated pod behavior and service routing.

4) **Frontend Development and Integration**
Built the React dashboard locally. Integrated all REST endpoints (upload, query, metrics) and tested full workflow end-to-end on the local Kubernetes cluster.

5) **EKS Deployment**
Provisioned an AWS EKS cluster with t3.medium node groups. Applied Kubernetes manifests, enabled Horizontal Pod Autoscaling based on CPU and memory thresholds, and deployed both backend and frontend.

- **Deployment:** used Kubernetes deployment scripts to deploy backend image with 2 initial replicas.
- **Service:** Configured a LoadBalancer service for traffic routing.
- **HPA:** set to autoscale between 2–6 replicas when CPU or memory utilization exceeds 70%.
- **Secrets:** stored sensitive credentials (S3, RDS, API keys) in Kubernetes Secrets.

6) **Performance Testing**
- **Load Levels:** 10, 50, 100, and 500 concurrent users via Apache JMeter.
- **Execution Flow:**
  a) Register
  b) Log in
  c) View existing expenses
  d) Upload receipt
  e) View updated expenses
- **Metrics Captured:** Ingestion latency, pod scaling events, and CPU/memory utilization.

7) **Monitoring Configuration**
Configured AWS CloudWatch dashboards to track pod CPU/memory metrics, request latencies, and error rates. Set alarms for threshold breaches to ensure operational visibility.

VIII. Evaluation

1) **Load Testing**
We used Apache JMeter to simulate loads ranging from 10 to 500 concurrent users, each executing the sequence: register → login → view bills → upload → view bills again. Below is the response time graph for 50 concurrent users.

Label	Req Count	Avg (ms)	Max (ms)	Throughput (req/s)
Bills	100	889	5 065	4.2
Register	50	4 779	10 418	3.8
Login	50	4 194	6 503	3.3
Upload	50	5 701	15 200	2.3

TABLE III
Key Load Test Metrics at 50 Concurrent Users

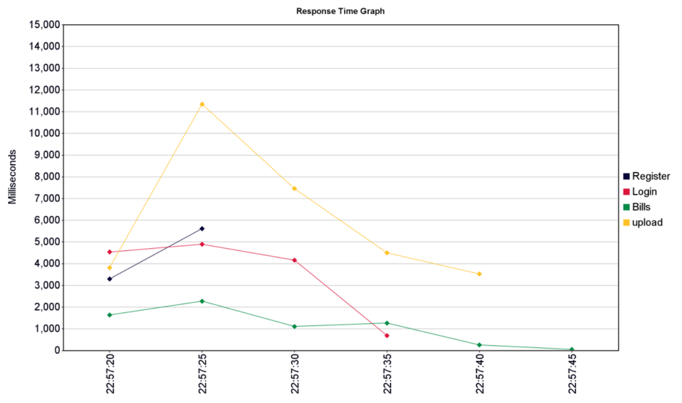

Fig. 2. Response time under 50-user load for each API endpoint: Register (navy), Login (red), Bills (green), Upload (yellow).

- Observations from the 50-user load test show that retrieving bills was the quickest operation, averaging 889ms per request and sustaining the highest throughput, since it performs a simple database query.
- User registration and login each averaged around 4–5s, reflecting the overhead of JWT handling and database writes.
- Receipt uploads were the most time-consuming, averaging 5.7s and peaking at 15.2s due to the OCR and LLM parsing steps before autoscaling kicked in.
- Overall, lighter-weight endpoints maintained higher throughput while compute-intensive tasks throttled performance.
- The Kubernetes Horizontal Pod Autoscaler proved effective at preventing sustained latency increases.
- Further improvements—such as adjusting scaling thresholds or introducing request buffering—could help smooth out the occasional upload spikes.

2) **Cluster Monitoring**

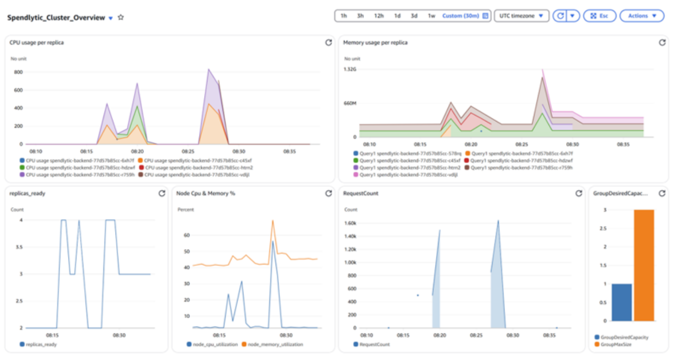

Fig. 3. Cluster monitoring: CPU, memory, replicas, requests, and scaling.

The AWS CloudWatch dashboard provides an at-a-glance view of our EKS cluster under load:

- **CPU Usage per Replica (top-left):** A stacked-area chart showing each pod's CPU consumption over time, with spikes up to 600–800mCPU during load tests.
- **Memory Usage per Replica (top-right):** A stacked-area chart showing pod memory usage rising to about 1.2GB under peak OCR+LLM processing.
- **Replicas Ready (middle-left):** A line chart indicating the number of running pods scaling from 2 up to 4 replicas when traffic increases, then dropping back as demand falls.
- **Node CPU & Memory% (middle-right):** Two overlaid lines tracking overall node utilization—CPU peaks near 60% and memory near 50%—which drive the autoscaler's decisions.
- **Request Count (bottom-left):** An area chart of incoming request rates, with two pronounced peaks around 1400–1600req/s corresponding to different load phases.
- **Desired vs. Max Capacity (bottom-right):** A bar chart comparing the EKS node-group's desired size (1 node) against its maximum allowed size (3 nodes), illustrating how scaling is bounded by the node-group configuration.

IX. Discussion and Next Steps

While our project achieved a reliable and scalable cloud-native expense tracking platform, several limitations and areas for future improvement emerged during development and evaluation. Below, we outline both technical and user-facing limitations observed during testing, followed by actionable next steps to evolve Spendlytic into a production-grade application.

1) **Limitations**

- **Latency under High Concurrency**
  Although our system autoscaled from 2 to 6 pods under 500 concurrent users, latency remained variable. Registration and login requests maintained low latency (4.5s), but upload requests (OCR + LLM parsing) peaked at 15.2s due to I/O bottlenecks and compute-bound parsing delays. Notably, the Horizontal Pod Autoscaler (HPA) exhibited a warm-up delay of approximately 30 seconds, during which traffic spikes caused temporary backlogs. Moreover, receipt uploads queued during this time experienced higher first-byte delay due to the sequential invocation of Textract and o4-mini, further exacerbating performance lags for users.
- **Instance Configuration**
  We deployed the EKS worker nodes using `t3.medium` CPU-based EC2 instances (2 vCPUs, 4 GB RAM) to balance cost and performance. While these instances handled moderate load effectively, they are not optimized for inference-heavy workloads. OCR and LLM parsing are inherently compute-intensive, and our testing showed average CPU utilization exceeding 70% during peak loads. GPU-based EC2 instances (e.g., `g4dn.xlarge`) would significantly accelerate LLM inference, reduce latency, and enable concurrent parsing, but were excluded due to budgetary and deployment complexity constraints. For long-term scalability, a hybrid CPU+GPU setup or serverless inference backend should be considered.
- **Mobile Accessibility**
  Currently, Spendlytic is implemented as a web-first platform. Although the frontend is mobile-responsive, it lacks native app integration. Most users capture receipts via smartphones, so the absence of native camera support limits real-time ingestion. Additionally, mobile OS features such as gallery access, push notifications for parsed results, and offline queuing are not available in our current web-based approach. As a result, usability for on-the-go scenarios remains constrained.
- **OCR Limitations**
  OCR quality remains highly sensitive to scan resolution, lighting, and font styles. AWS Textract produced accurate results for clean, machine-printed receipts but struggled with low-quality images, handwritten notes, or receipts with graphical backgrounds. In some cases, subtotal and tax values were misinterpreted as item prices. o4-mini, while lightweight and fast, showed limitations in parsing nested or multi-line tabular data, such as itemized grocery lists. These errors propagate downstream, affecting chart accuracy and budget summaries.
- **Privacy and Data Retention**
  From a user privacy perspective, receipts may contain sensitive data (e.g., medication, alcohol, health services, or geo-location clues). While our platform employs HTTPS communication, JWT authentication, and encrypted S3 + RDS storage with IAM-based access control, users currently lack visibility or control over their stored data. There is no built-in mechanism for deleting past uploads, viewing access history, or redacting personally identifiable information (PII). This gap poses concerns for compliance with data protection frameworks such as GDPR or CCPA, where user consent and data lifecycle transparency are critical.

2) **Next Steps**

- **Mobile App and Camera Upload**

We plan to develop a React Native-based mobile app that enables real-time receipt capture using device cameras. This will enhance usability for on-the-go users, offer native camera integration, and allow previews of parsed metadata before submission. Offline mode with local caching and background uploads will further improve user experience.

- **End-to-End Encryption & Privacy Controls**
  To elevate data protection, we aim to introduce client-side encryption using libraries such as AWS KMS or WebCrypto API. This will ensure that sensitive receipt content is encrypted prior to transmission. Additionally, users will gain access to a privacy dashboard where they can:
  a) Delete stored receipts on demand
  b) Redact specific fields (e.g., vendor name, items)
  c) View data access history and logs

- **Intelligent OCR & LLM Routing**
  Future versions will support intelligent model routing. For instance, if AWS Textract yields low confidence, the pipeline will switch to LayoutLMv3 or a fine-tuned Gemini Flash model using semantic thresholds. We also plan to introduce optional user confirmation dialogs when extracted data exceeds predefined ambiguity thresholds—e.g., missing total or unreadable dates.

- **Real-Time Analytics Dashboard**
  While our current dashboard supports aggregate receipt views, we intend to incorporate:
  a) Monthly budget forecasting using linear regression or Prophet
  b) Anomaly detection for sudden expense spikes
  c) Budget goal setting and category-based savings suggestions

  This transforms Spendlytic from a passive tracker into a proactive financial assistant.

- **Improved Latency Handling**
  To reduce spike-based delays in parsing, we will explore:
  a) Provisioned concurrency for OCR Lambda functions to eliminate cold starts
  b) Scheduled pod pre-warming (e.g., via cronjobs) during expected traffic hours
  c) Asynchronous parsing via Amazon SQS queues and Lambda consumers for decoupled execution

- **Enhanced Cloud Monitoring**
  We plan to enhance observability by integrating Amazon Managed Grafana with CloudWatch. Advanced metrics such as request throttling, S3 I/O time, and RDS query performance will be visualized in real-time. Alerts based on percentile-based latency and resource usage trends will allow proactive tuning. Cost visibility dashboards will also be added to manage operational budgets.

- **Database Optimization**
  PostgreSQL performed reliably, but we noted increased response time for queries involving JSON fields at scale. To address this, we will:
  a) Create GIN indices on JSONB fields
  b) Offload analytical queries to Amazon Redshift or Athena
  c) Cache frequent reads using Redis for hot endpoints

  For extreme-scale workloads, migrating expense summaries to DynamoDB may be evaluated for low-latency access.

- **Production-Grade Deployment**
  Currently, our deployment uses basic Kubernetes manifests and provisions two replicas per service. Future upgrades will include:
  a) Multi-AZ RDS deployment for failover resilience
  b) CloudFront CDN caching for static assets
  c) Canary deployments with health checks and gradual rollout for backend/React updates

In summary, Spendlytic lays a strong foundation for scalable, cloud-native personal expense tracking. Our next milestones will target production readiness, mobile usability, intelligent parsing, and advanced user controls—bringing us closer to a privacy-aware, analytics-driven, and user-first financial assistant.

Acknowledgments

We thank the professor and TAs for their support and timely feedback throughout the project. Their guidance was instrumental in shaping the technical direction and ensuring the successful delivery of our cloud-native application.