SMART BUDGET TRACKER

Angad Deep Singh
University of Massachusetts Amherst
Department of ECE
MA 01375
asingh@umass.edu

Poojitha Umashnkar Singh
University of Massachusetts Amherst
Dept of ECE
MA 01375
poojithaumas@umass.edu

Robert Lewandowski
University of Massachusetts Amherst
Department of ECE
MA 01102
rlewandowski@umass.edu

Deepikaveni Venkatanarayanan
University of Massachusetts Amherst
Department of ECE
MA 01102
dvenkatanara@umass.edu

ABSTRACT

Budgeting money will always be a consumer need, as the cost of living can be a difficult task to manage for the average American. Financial technology is continuously expanding and securing transactions and enhancing consumer needs are still critical challenges. This project leverages Plaid APIs to create a budgeting solution and financial tool to analyze and categorize users' transaction data, displaying insights into the users spending habits. Budget tracking and financial awareness can seem overwhelming and tedious, however, our solutions provide an automatic classifying and tracking of transactions into typical budgeting categories like entertainment, groceries, and transportation. By processing transaction data, the application generates spending reports that allow users to easily analyze their monthly spendings. This prototype provides a proof of concept for a functional, cloud based financial tracker service.

Keywords

AWS; EC2; DynamoDB; S3; Plaid API; Flask; Budget Tracker; Cloud Computing; Serverless Architecture

1. INTRODUCTION

The rise of Fintech has made consumer transaction management easy, but demands robust tools to enhance the processing and financial visibility at a larger scale. While many solutions focus more on fraud detection, our solutions address a lifestyle need, personalized budget tracking with automatic transaction analysis. Due to the sensitive nature of our project, and our lack of focus on security, we decided against using real banking data and developed a prototype using Plaid's Sandbox Environment to simulate bank transactions for our dataset. The sandbox environment provided us the freedom to securely test our solution without the threat of compromising real banking data.

Our system fetches synthetic transaction data via Plaid's API sandbox, processes it through a cloud-based pipeline that we built on AWS, then delivers the synthetic transactions via a frontend web interface. The backend was developed using Python/Flask and handles the transaction fetching and cleaning of the transactions, while AWS EC2 serves as the application. The transaction data is stored using DynamoDB, a cloud tool great for scalable data retrieval, and intermediate datasets are archived in S3. The frontend provides the users a digestible, graphical representation of their spending patterns, making budget tracking very easy.

This approach shows how Fintech APIs, like Plaid, and cloud infrastructure, like AWS, can be utilized to build practical and scalable financial tools. This prototype visualizes how a cloud pipeline could be integrated using real transaction data with the proper security protocols, or even machine learning with a much larger sample size of real transaction data.

2. SYSTEM SETUP AND INFRASTRUCTURE

To handle all backend processing for budget tracking we created an **AWS** infrastructure that includes resources like storage, database systems, and secure access controls. We started by creating an AWS account and configuring an Identity and Access Management (IAM) role . We utilized a t2.micro EC2 instance to host the Flask backend server and execute Python scripts for creating access tokens, link tokens, fetching transactions data, etc. Furthermore, we implemented a tiered storage approach with both an S3 bucket , for storing raw JSON archives of transaction data, and a Dynamodb table to structure and categorize the user transactions. Custom IAM roles granted us the ability to allow the EC2 instance to access the S3 and DynamoDB.

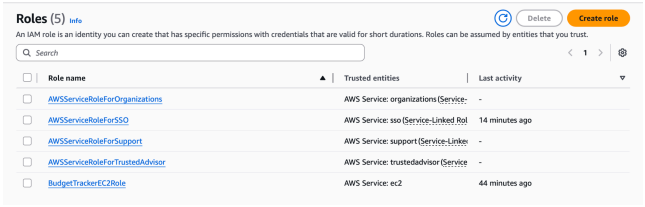


Fig. 1 IAM roles granting EC2 instance permissions to access S3 and DynamoDB

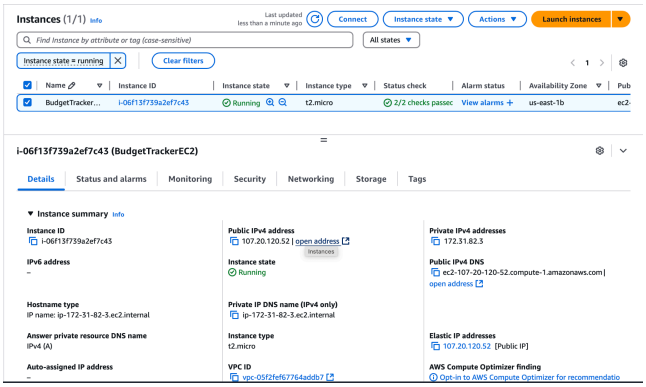


Fig. 2 EC2 instance showing successful connection and running environment

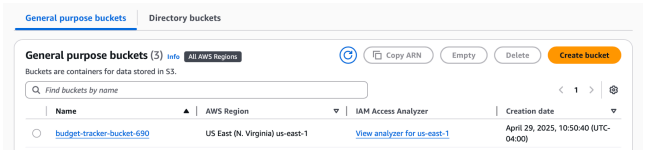


Fig 3. S3 Bucket "budget-tracker-bucket-690" is integrated with EC2 instance

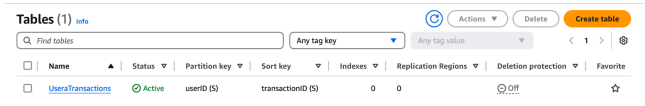


Fig. 4 DynamoDB table "user-transactions" integrated with EC2 instance

3. IMPLEMENTATION

3.1 Plaid API Integration

A primary part of the cloud based budget tracker was designing and implementing the Plaid API integration, which serves as the base for fetching and processing transaction data. We created a developer account on Plaid's dashboard, refer to figure 5, to obtain credentials for the sandbox environment, which includes the ClientID and Sandbox Secret Key, allowing secure access to Plaid's testing environment. Utilizing Plaid's Quickstart GitHub repository [2] allowed us to set up an instance of their frontend and backend locally on a machine. Within this local sandbox environment a few fake plaid banks, simulated financial institutions, generate an access token for receiving synthetic transaction data. In our case, we accessed Plaidypus, to generate the required access token. This step is important, as it allowed us to begin working within the AWS environment.

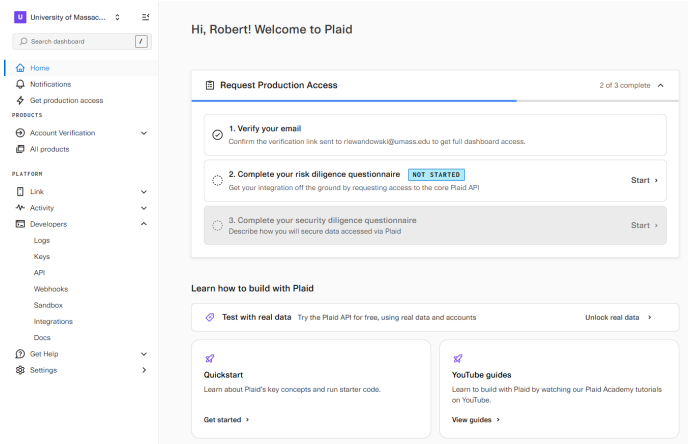


Fig 5. Plaid Developer Dashboard with Sandbox access enabled

However, a challenge we experienced in this process was difficulty launching the quickstart instances locally on a Windows machine.

Windows machines require the *core.symlinks plaid_quickstart* GitHub [2], since the repository makes use of symbolic links. This came as a challenge, since even with symbolic links enabled on the machine and using the required symbolic setup, the machine was unable to create the quick launch instance that enables the generation of the access token. However, simply creating the instance on a macOS allowed us to quickly access the instance and create our access token. Refer to figure 6 for clientID, secret key, and sandbox environment.

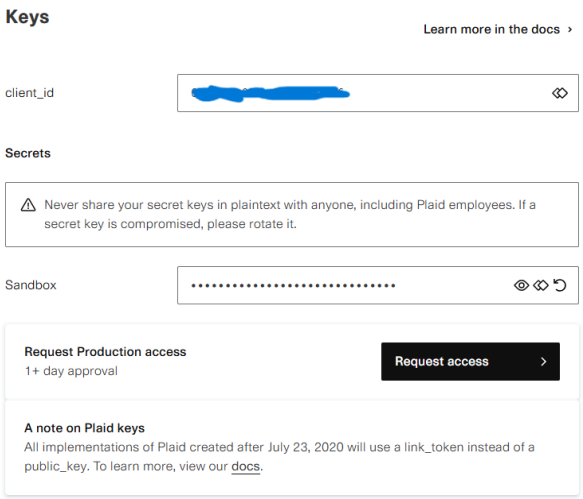


Fig 6. Blurred Client ID and Secret Setup

3.2 Backend API Development with Flask

The next step was to develop and deploy a *plaid_api.py* on the AWS EC2 instance, serving as the main part of the cloud, Plaid integration. Utilizing plaid's documentation [1], we were able to create the main functions. The three main functionalities are handled within *plaid_api.py*: generating link tokens, exchanging public tokens, and fetching the fake transaction data.

The backend needed to access our plaid sandbox credentials, PLAID_CLIENT_ID and PLAID_SECRET, which was loaded via python-dotenv for security (Exposing the clientID and secret directly in the code could lead to vulnerabilities). A *create_link_token* was created to make requests under specific parameters: application name, Smart Budget Tracker; plaid product scopes, transactions in our case; and finally regional settings, US code. By creating link tokens it enabled connection to the frontend.

To introduce authentication, we created *exchange_public_token, which* swaps temporary public tokens for permanent access tokens.

Once the permanent access tokens are received they are stored and utilized in *get_transactions* to retrieve the fake sandbox transaction data within the specified date ranges. For prototype purposes we used the date range 2023-01-01 to 2023-12-31.

Deployed on EC2 with Gunicorn, this service formed the secure bridge between Plaid's API and our AWS infrastructure.

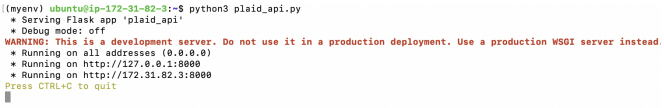


Fig 7. Flask server running successfully on EC2 terminal

```
@app.route('/create_link_token', methods=['GET'])
def create_link_token():
    request_data = LinkTokenCreateRequest(
        user=LinkTokenCreateRequestUser(client_user_id="user-123"),
        client_name="Smart Budget Tracker",
        products=[Products("transactions")],
        country_codes=[CountryCode('US')],
        language="en"
    )
    response = client.link_token_create(request_data)
    return jsonify(response.to_dict())

@app.route('/exchange_public_token', methods=['POST'])
def exchange_public_token():
    public_token = request.json.get('public_token')
    try:
        response = client.item_public_token_exchange({'public_token': public_token})
        access_token = response['access_token']
        return jsonify({'access_token': access_token})
    except Exception as e:
        return jsonify({'error': str(e)}), 400


@app.route('/get_transactions', methods=['POST'])
def get_transactions():
    data = request.get_json()
    access_token = data.get('access_token')

    try:
        # Set date range
        start_date = "2023-01-01"
        end_date = "2023-12-31"

        response = client.transactions_get({
            'access_token': access_token,
            'start_date': start_date,
            'end_date': end_date
        })

        return jsonify(response.to_dict())
    except Exception as e:
        return jsonify({'error': str(e)}), 400

if __name__ == '__main__':
    app.run(host="0.0.0.0", port=8000)
```

Fig. 8 Nano editor displaying Flask routes: create_link_token, exhange_public_token, and get_transactions

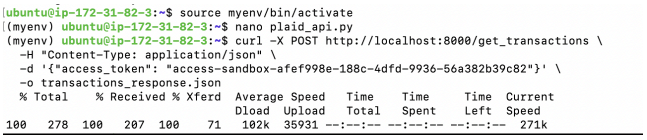


Fig. 9 Curl commands showing token exchange and transaction fetch requests

3.3 Fetching Data from Plaid API

After getting a year's worth of fake transactions data via Plaid API, we implemented a dual-storage that ensures that raw data is archived and efficiently queried. The JSON file contains all transaction details in its raw form, timestamped with a date and time, and archived in the S3 bucket *budget-tracker-bucket-690* in case a backup would be required. In parallel, we parse the JSON file to extract key front end required data, like cost amount, merchant name (uber, McDonald's, etc.), category (groceries, transportation, entertainment, etc.), and date of purchase. This is done using the *store_transactions.py* Python script, which automates the entire process of inserting the data into the Dynamodb table *UserTransaction*. The entire workflow can be referred to in figure 10, the JSON payload preview; figure 11, S3 backups; and figure 12, Dynamodb's transaction records.

Fig. 10 Preview of fetched transaction JSON using cat transactions_response.json

Fig. 11 Data fetched successfully and uploaded to S3 and DynamoDB

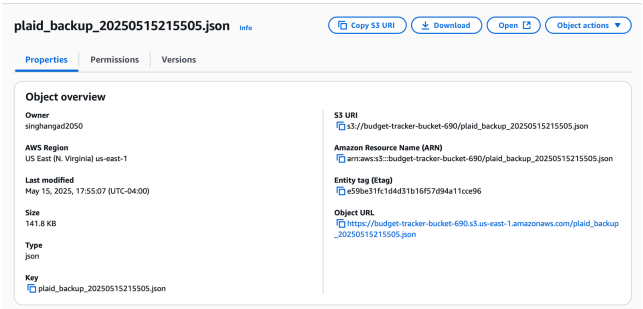


Fig. 12 S3 bucket showing successfully uploaded backup JSON file

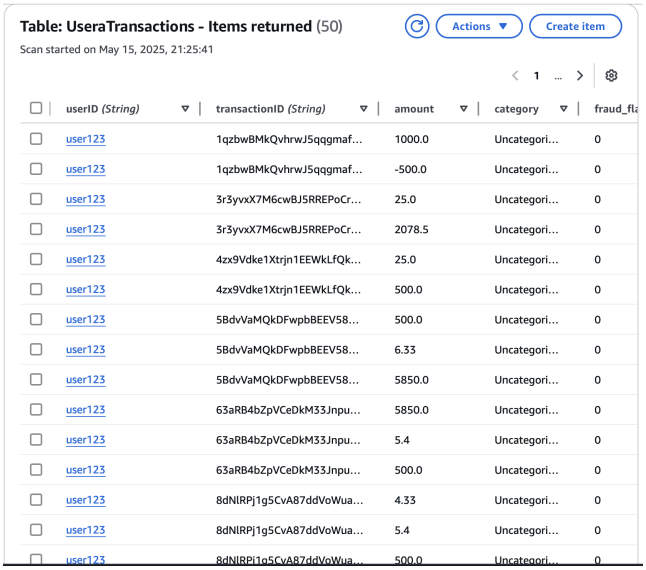


Fig. 13 DynamoDB rows reflecting inserted transactions

3.4 Data Cleaning and Categorisation

To evaluate the efficacy of our backend infrastructure and the integration of Plaid transaction data, we implemented data storage and retrieval through Amazon DynamoDB. Each financial transaction—whether retrieved from the Plaid API or uploaded via CSV—is structured and stored in a dedicated table named user transaction .

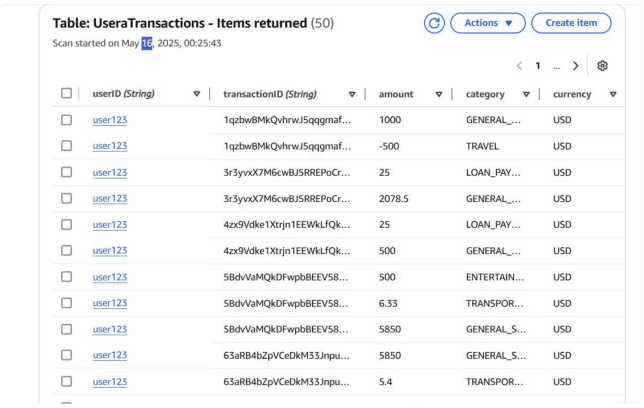


Fig 14: Categorised data in Dynamo DB

Each record in the UseraTransactions table consists of structured attributes that enable seamless financial data tracking and analysis. The userID field uniquely identifies the user associated with each transaction. The transactionID serves as a distinct identifier or hash for every transaction entry, ensuring uniqueness and traceability. The amount field captures the transaction's monetary value, where positive values indicate credits (income) and negative values represent debits (expenses). The category field classifies the transaction into meaningful types such as GENERAL, TRAVEL, LOAN_PAYMENT, ENTERTAINMENT, or TRANSPORT, facilitating categorization and budget analysis. Lastly, the currency field denotes the transaction currency, which is standardized to USD in the current implementation.

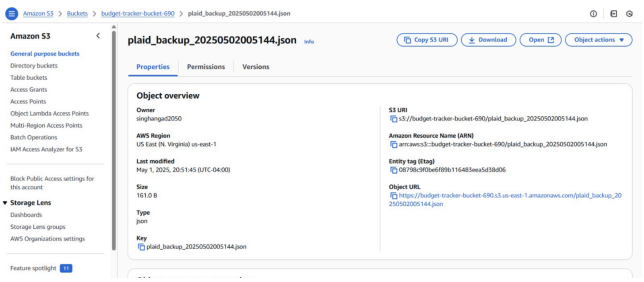


Fig 15 Json file backup in S3

This `.json` file likely contains a backup of Plaid transaction data, which may include account metadata, transaction history, and user spending summaries. Such a backup is particularly useful for restoring the application state in case of data loss, enabling offline processing of financial records, and feeding a Flask backend to simulate or serve mock transaction APIs, as well as supporting budget analysis workflows. Additionally, maintaining periodic backups in JSON format allows for version control, easier debugging during development, and seamless integration with cloud-based storage and analytics pipelines. It also ensures data portability and consistency across environments, especially when testing new features or migrating services.

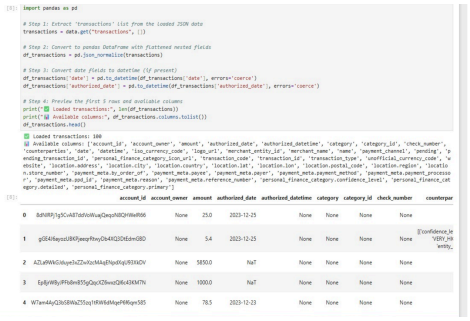


Fig 16 Categorisation of the data

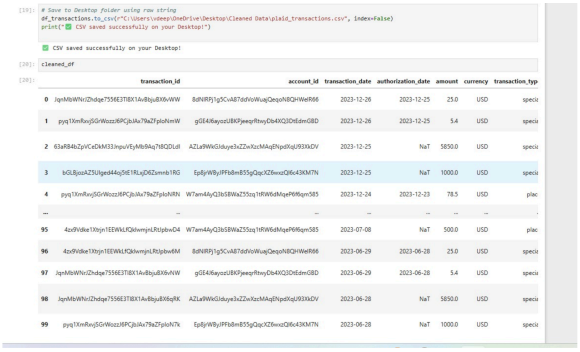


Fig 17 Cleaned Data including key fields

As part of the final stage in the data cleaning process, the refined transaction dataset was saved locally as a structured CSV file. The cleaned data includes key fields such as transaction ID, account ID, transaction date, authorisation date, amount, currency, and transaction type. Unnecessary or incomplete fields were removed, and datetime values were standardised to a consistent format. The output confirms that each transaction is now uniformly structured and free from nested or missing attributes. This exported file serves as a reliable source for subsequent visualisation, dashboard rendering, or database insertion into services like Amazon S3 or Dynamodb. It also ensures reproducibility and facilitates integration with third-party financial analysis tools.

All data cleaning, transformation, and preprocessing were carried out within a Jupyter Notebook environment, enabling interactive development and seamless data inspection. The notebook was used to load Plaid's raw transaction JSON, normalise nested

fields, convert date formats, parse counterparty details, handle missing values, and export a cleaned CSV file. The final step involved uploading this cleaned dataset to Amazon DynamoDB. A connection to the Dynamodb service was established using AWS credentials, targeting the `UserTransactions` table. Each transaction was processed row by row, where key attributes such as user ID, transaction ID, amount, date, merchant name, and category were extracted or defaulted when missing. Optional fields like currency and payment channel were conditionally included based on availability. Each item was structured as a dictionary and inserted into Dynamodb using the `put_item` operation, with exception handling in place to log any upload errors. This end-to-end pipeline ensured that structured, cleaned, and cloud-stored financial data was readily available for use in backend APIS, analytics, and frontend dashboard visualisations.

```
[27]: # Initialize DynamoDB resource
from decimal import Decimal
dynamodb = boto3.resource(
    'dynamodb',
    region_name='us-east-1',  # or your actual AWS region
    aws_access_key_id='AKIA3G7PVKQCUAHOLMEE',
    aws_secret_access_key='lhsmId6GdU7xafEUBeXVhvOUC6kgoif8VjYsQuFI7'
)

table = dynamodb.Table('UseraTransactions')  # <-- change if your table name is different

# Add a static user ID (or dynamically assign based on logic)
user_id = 'user123'

# Upload each transaction
for _, row in df_transactions.iterrows():
    try:
        item = {
            'userID': user_id,
            'transactionID': str(row['transaction_id']),
            'amount': Decimal(str(row['amount'])),
            'merchant': row.get('merchant_name', 'Unknown') or 'Unknown',
            'category': row.get('category_primary', 'Uncategorized') or 'Uncategorized',
            'date': str(row['transaction_date']),
        }

        # Optional fields (check if they exist in your CSV)
        if 'payment_channel' in row:
            item['payment_channel'] = row['payment_channel']
        if 'currency' in row:
            item['currency'] = row['currency']

        table.put_item(Item=item)

    except Exception as e:
        print(f"❌ Error inserting transaction {row['transaction_id']}: {e}")

print("✅ All transactions inserted into DynamoDB.")
```

Fig 18: Storing all the data into Dynamo DB

4. FRONTEND IMPLEMENTATION

To facilitate secure and user-friendly bank integration, the application uses Plaid Link, a client-side JavaScript module provided by Plaid. This component is embedded directly into the frontend using a simple script snippet. When users click the "Link Bank Account" button on the dashboard, Plaid Link launches a secure modal interface where users can search for and select their financial institution. After logging in through Plaid's encrypted environment, the user's bank account is securely connected. Once the connection is successful, Plaid Link returns a public_token to the frontend. This token represents the linked account and is temporarily stored on the client. It is then sent to the backend server to be exchanged for a permanent access_token, which is used to securely access the user's transaction history and account metadata. This seamless and secure flow eliminates the need for handling sensitive banking credentials directly and enables fast integration of real-time financial data into the dashboard.

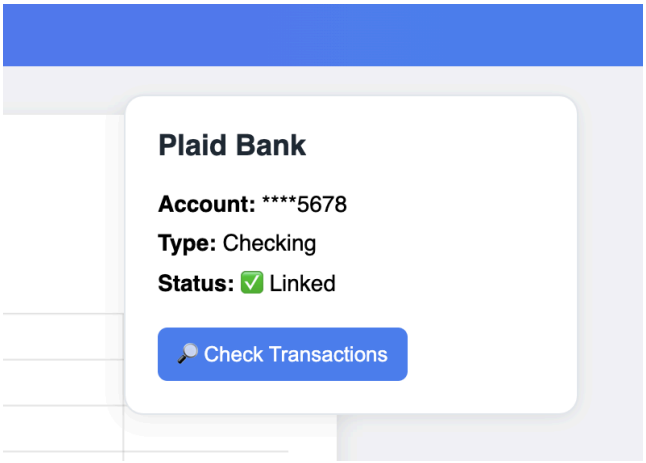


Fig 19 Linking the Bank account

public_token request.json['public_token access_token =
client.Item.public_token.exchange(public_token)['access_token']

return jsonify({'access_token': access_tokenAfter the bank account is successfully linked through Plaid Link, the frontend sends the returned public_token to a Flask backend API endpoint over a secure connection. The backend then communicates with the Plaid API to exchange the temporary public_token for a permanent access_token. This access_token is securely stored and is used to retrieve the user's financial data, including transaction history, account balances, and account details. This process ensures that sensitive authentication remains securely handled by the backend, while the frontend is only responsible for initiating the flow and presenting the insights to the user.

from flask import Flask, request, jsonify

from plaid import Client

app = Flask(__name__)

client=Client(client_id='******),
environment='sandbox')

@app.route('/exchange', methods=['POST'])

def exchange_token():

})

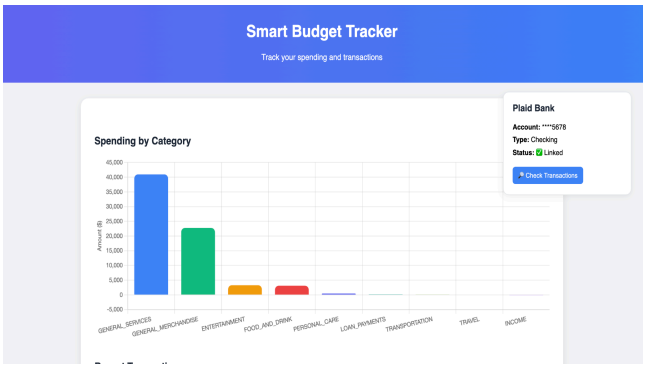


Fig 20 Front end Transaction

Once the transaction data is fetched—either directly from the backend or retrieved from a database such as Amazon DynamoDB—it is processed and displayed in a clean, interactive frontend using the Chart.js library and structured HTML tables. This allows for intuitive financial insights without overwhelming the user with raw data.

In the first screenshot, we observe the **S**pending by Category bar chart. This visualisation serves as the primary analytical tool of the dashboard, offering a high-level snapshot of where the user's money is going. Categories such as General Services, Merchandise, Entertainment, Food and Drink, Personal Care, and Transportation are plotted along the x-axis, while the corresponding amount spent in each category is displayed on the y-axis. Each bar is colour-coded for easy distinction and aesthetic clarity.

The purpose of this chart is to allow users to quickly identify their largest spending areas, evaluate whether spending aligns with their budgeting goals, and spot irregularities such as an unexpected spike in a low-priority category. For example, if "Entertainment" spending far exceeds others in a short time frame, the user may consider adjusting their behaviour or setting alerts.

The chart updates automatically based on incoming transaction data from linked bank accounts via the Plaid API. As new transactions are pulled in and categorised, the visualisation re-renders to reflect the latest financial activity. This ensures that users always have access to **r**eal-time financial insights—without needing to refresh or manually filter through records.

Recent Transactions

Transaction ID	Date	Amount ($)
TXN1001	2024-10-15	1200.00
TXN1002	2024-10-14	75.50
TXN1003	2024-10-13	45.75
TXN1004	2024-10-12	220.00
TXN1005	2024-10-11	30.99

Transaction Summary

Date	Amount ($)	Category
2024-10-15	1200.00	ENTERTAINMENT
2024-10-14	75.50	FOOD_AND_DRINK
2024-10-13	45.75	TRANSPORTATION
2024-10-12	220.00	LOAN_PAYMENTS
2024-10-11	30.99	GENERAL_MERCHANDISE

Fig 21 Transaction Data

The Fig 21 shows the **t**ransaction history and summary tables, located below the spending chart. The "Recent Transactions" table lists individual transactions with their ID, date, and amount, allowing users to quickly track recent spending and verify financial activity. This helps in identifying recent purchases and spotting any suspicious charges.

Below it, the Transaction Summary table groups transactions by date and category, displaying the total amount spent per category each day. This offers users a clearer view of spending patterns, helping them recognize where money is going daily. For example, repeated high expenses in categories like "Entertainment" or "Food and Drink" may indicate potential areas for budget adjustments.

Both tables are styled for readability and designed to display dynamic data from Plaid or backend storage like DynamoDB. Together, they provide both detailed and summarized financial insights, making it easier for users to monitor and manage their finances effectively.

```
<script>
  const ctx = document.getElementById("categoryChart").getContext("2d");
  new Chart(ctx, {
    type: "bar",
    data: {
      labels: [
        "GENERAL_SERVICES", "GENERAL_MERCHANDISE", "ENTERTAINMENT",
        "FOOD_AND_DRINK", "PERSONAL_CARE", "LOAN_PAYMENTS",
        "TRANSPORTATION", "TRAVEL", "INCOME"
      ],
      datasets: [{
        label: "Total Spending ($)",
        data: [40950, 22739, 3268, 3097, 471, 175, 75, 0, -25],
        backgroundColor: [
          "#3b82f6", "#10b981", "#f59e0b", "#ef4444",
          "#6366f1", "#14b8a6", "#84cc16", "#f97316", "#8b5cf6"
        ],
        borderRadius: 8
      }]
    },
    options: {
      responsive: true,
      plugins: {
        legend: { display: false },
        tooltip: {
```

Fig 22 Snippets of the front end code

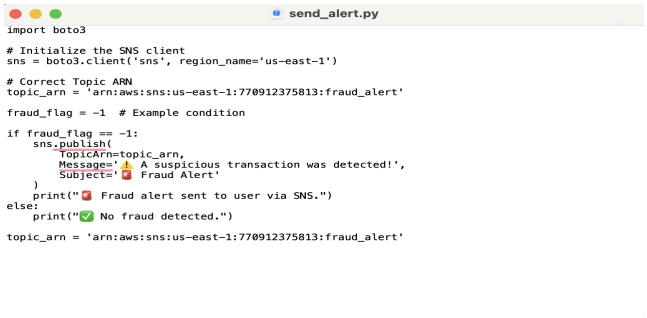
The Amazon S3 interface where the Smart Budget Tracker frontend files were uploaded. The S3 bucket is named budget-tracker-frontend1 and is used to host the static HTML files for the project's user interface. Two files are present in the bucket: index.html, which is the main dashboard page, and indexpooj.html, which appears to be a backup or test version.

To make the dashboard publicly accessible, static website hosting was enabled in the bucket's Properties tab, and index.html was set as the entry point. A bucket policy was applied to allow public read access to the HTML files.After configuration, AWS provides a public website URL. By opening this link in any web browser, users can view the dashboard without needing a web serve

5. FRAUD ALERT WITH SNS

This Python script implements an automated fraud alert mechanism using Amazon SNS (Simple Notification Service). The core purpose of this setup is to notify users immediately when a suspicious transaction is detected in their financial activity. The script begins by importing the boto3 library, which is the official AWS SDK for Python and is used to interact with AWS services. An SNS client is then initialized for the us-east-1 region, matching the region where the SNS topic was created.

A specific SNS topic ARN (arn:aws:sns:us-east-1:770912375813:fraud_alert) is defined to direct where the alert messages will be published. This ARN uniquely identifies the SNS topic that subscribers (such as email addresses or phone numbers) have opted into receiving notifications from. A simulated fraud_flag is set to -1, representing a detected fraudulent transaction. The script then checks this flag — if the flag is set to -1, it publishes an alert message to the SNS topic with a clear subject and body indicating the detection of suspicious activity. Any user subscribed to this SNS topic will instantly receive this message via their chosen channel (email or SMS). If no fraud is detected, the script simply logs a safe message to the console.

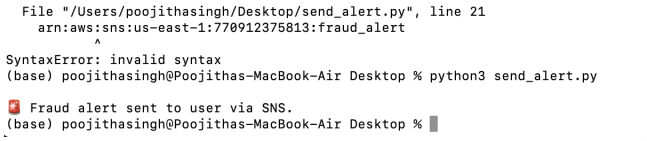


send_alert.py

```
import boto3

# Initialize the SNS client
sns = boto3.client('sns', region_name='us-east-1')

# Correct Topic ARN
topic_arn = 'arn:aws:sns:us-east-1:770912375813:fraud_alert'

fraud_flag = -1  # Example condition

if fraud_flag == -1:
    sns.publish(
        TopicArn=topic_arn,
        Message='⚠️ A suspicious transaction was detected!',
        Subject='🚨 Fraud Alert'
    )
    print("🚨 Fraud alert sent to user via SNS.")
else:
    print("✅ No fraud detected.")

topic_arn = 'arn:aws:sns:us-east-1:770912375813:fraud_alert'
```

Fig 23 Python code for SNS alert

This message was delivered to all users subscribed to the topic via their chosen communication method (such as email or SMS). Additionally, a confirmation message was printed in the terminal to indicate that the fraud alert had been sent successfully, verifying that the script and SNS integration were functioning as intended.



```
  File "/Users/poojithasingh/Desktop/send_alert.py", line 21
    arn:aws:sns:us-east-1:770912375813:fraud_alert
          ^
SyntaxError: invalid syntax
(base) poojithasingh@Poojithas-MacBook-Air Desktop % python3 send_alert.py

🚨 Fraud alert sent to user via SNS.
(base) poojithasingh@Poojithas-MacBook-Air Desktop %
```

Fig 24 The SNS sends an alert to the user

The email shown in the screenshot was successfully delivered to poojithumas@umass.edu as part of the fraud alert system powered by Amazon SNS. The subject line of the email, "Fraud Alert", was configured in the Python script using the Subject parameter of the sns.publish() method. The sender name, displayed as "Topic Arn", is derived from the default topic configuration in SNS, which labels the sender using the topic metadata. The timestamp 5:29 A" marks the exact moment the script detected a suspicious transaction (via

fraud_flag == -1) and triggered the alert. The body of the email contains the warning message "A suspicious transaction was detected!", which was also defined in the script using the Message parameter. Additionally, AWS SNS automatically appends an unsubscribe link at the bottom of the email, allowing the recipient to opt out of future notifications from this topic. This complete email notification confirms that the backend alert logic and the SNS delivery mechanism are functioning end-to-end as intended.

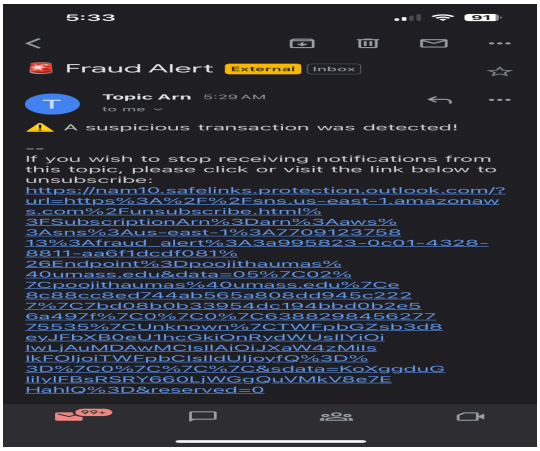


Fig 25 Fraud Alert sent to mail

6. DASHBOARD AND VISUALISATION

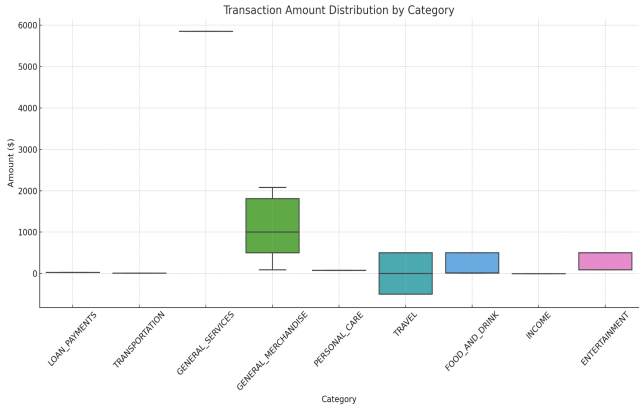


Figure 26 dashboard 1

The box plot titled "Transaction Amount Distribution by Category" illustrates how transaction amounts vary across different spending categories. Each box shows the middle 50% of transaction values, with the line inside indicating the median. Categories like General Services and General Merchandise display wider spreads and higher transaction amounts, suggesting more variable and larger expenses. In contrast, categories like Loan Payments and Transportation show tighter, more consistent spending. This visualization helps identify outliers and highlights which categories may require closer financial oversight.

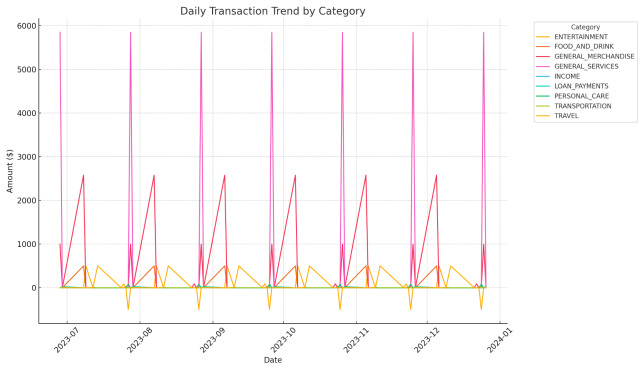


Figure 27 Dashboard 2

The line graph titled "Daily Transaction Trend by Category" shows how spending varies over time across multiple transaction categories. Each line represents a different category, tracking daily amounts throughout the year. Categories like General Services and General Merchandise show consistently high spikes, indicating recurring large expenses. Others, like Entertainment and Food and Drink, have moderate fluctuations, while some categories like Transportation, remain relatively flat. This visualisation is useful for spotting spending patterns, detecting anomalies, and understanding seasonal or monthly financial behaviours.

7. CONCLUSION

This smart budget tracking system presents a comprehensive approach to personal finance management by combining real-time analytics, category-wise visualization, and secure alerting mechanisms. Through intuitive dashboards built with HTML, CSS, and Chart.js, users can easily track spending behavior across key categories like Entertainment, Travel, Groceries, and Loan Payments. The system processes raw CSV transaction data to dynamically generate visualizations, including bar

charts, pie charts, line graphs, and box plots, enabling clear comparisons and trend analysis over time.

Additionally, the integration with Amazon Web Services (AWS) plays a critical role in automation and security. Transaction data is uploaded to Amazon S3 for scalable storage, while Amazon SNS (Simple Notification Service) is used to send fraud alerts via email or SMS whenever anomalies are detected (e.g., unusually high spending). This proactive notification system ensures that users stay informed and can take immediate action.

The modular structure—with backend processing using Python and frontend deployment on AWS—makes the system both flexible and extendable. Users can filter data, check bank linkage (via mock Plaid UI), and visualize insights without needing deep technical knowledge. Ultimately, this project demonstrates how data, cloud technologies, and responsive design can be combined to deliver a secure, user-friendly budgeting tool that enhances both awareness and control over financial activities.

8. REFERENCES

[1] Ahmed, M., Mahmood, A. N., & Hu, J. (2016). A survey of network anomaly detection techniques. *Journal of Network and Computer Applications*, 60, 19–31. https://doi.org/10.1016/j.jnca.2015.11.016

[2] Wärnestål, P. (2011). Designing personal financial management tools for everyday life: A field study of in-home budgeting. In *Proceedings of the 2011 IEEE International Conference on Consumer Electronics (ICCE)* (pp. 365–366). Las Vegas, NV, USA. https://doi.org/10.1109/ICCE.2011.5722676

[3] Zhang, Y., Wang, Q., & Wang, Y. (2020). A personal finance management system based on machine learning and visualization. In *2020 5th International Conference on Intelligent Computing and Signal Processing (ICSP)* (pp. 424–429). IEEE. https://doi.org/10.1109/ICSP51882.2020.9321035

[4] Nguyen, T. T., & Armitage, G. (2008). A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys & Tutorials*, 10(4), 56–76. https://doi.org/10.1109/SURV.2008.080406

[5] Kim, Y., Lee, J., & Kim, K. (2019). Real-time fraud detection using stream mining and anomaly detection. *Procedia Computer Science*, 151, 927–932. https://doi.org/10.1016/j.procs.2019.04.129