

FINAL ITERATION



THE UNIVERSITY OF TEXAS AT ARLINGTON

**Advanced Topics in Software
Engineering CSE 6324 – Section 001**

Team 07

Final Iteration

**(Written
Deliverable)**

Team 7:

SAI SRI LAKSHMI NANNAPANENI - 1002077755

AMAN MITTAL - 1001979035

PAVAN SAI AKULA - 1002082165

RUKESH SAI KASTURI - 1002049844

Git_Hub Link: https://github.com/mittal-aman7/brownie_ase

GAS USAGE ANOMALY DETECTOR

Implementation Plan for Anomaly Detection in Gas Usage

The objective of this detector is to develop and deploy an anomaly detection system for gas usage in smart contracts on the Ethereum blockchain. The system aims to provide early warning of irregular patterns, potentially indicative of inefficiencies or security concerns. This report outlines a phased implementation plan designed to minimize project risk and maximize user value, along with an overview of the project's differentiation from related applications and strategies to mitigate the top identified risks.

In the iteration 2 we have worked on Database setup , data collection phases and partially completed anomaly detection part. In this Final Iteration we made few changes for the last work we did and worked on Initialization of the Data Repository , Captured Transaction Data Identified Outliers in Gas Usage and Automated Detection and Notification

Phase 1: Set Up the Database [1]

```
import psycopg2
from psycopg2 import sql, DatabaseError

DATABASE_NAME = 'gas_usage'
USER = 'Aman_Mittal'
PASSWORD = 'Password@123'
HOST = 'localhost'

# Function to create the current_transactions table
def create_current_transactions_table():
    try:
        with psycopg2.connect(dbname=DATABASE_NAME, user=USER, password=PASSWORD, host=HOST) as conn:
            with conn.cursor() as cur:
                cur.execute(sql.SQL('''
                    CREATE TABLE IF NOT EXISTS current_transactions (
                        id SERIAL PRIMARY KEY,
                        tx_hash TEXT NOT NULL,
                        gas_used INTEGER NOT NULL,
                        timestamp TIMESTAMP NOT NULL
                    )
                '''))
    except (Exception, DatabaseError) as e:
        print(f"Database error occurred: {e}")

# Function to create the historical_transactions table
def create_historical_transactions_table():
    try:
        with psycopg2.connect(dbname=DATABASE_NAME, user=USER, password=PASSWORD, host=HOST) as conn:
            with conn.cursor() as cur:
                cur.execute(sql.SQL('''
                    CREATE TABLE IF NOT EXISTS historical_transactions (
                        id SERIAL PRIMARY KEY,
                        tx_hash TEXT NOT NULL,
                        gas_used INTEGER NOT NULL,
                        timestamp TIMESTAMP NOT NULL
                    )
                '''))
    except (Exception, DatabaseError) as e:
        print(f"Database error occurred: {e}")

# Function to insert transaction data into current_transactions
def insert_current_transaction(tx_hash, gas_used, timestamp):
    try:
        with psycopg2.connect(dbname=DATABASE_NAME, user=USER, password=PASSWORD, host=HOST) as conn:
            with conn.cursor() as cur:
                cur.execute(sql.SQL('''
                    INSERT INTO current_transactions (tx_hash, gas_used, timestamp)
                    VALUES (%s, %s, %s)
                '''))
```

```

def store_historical_transactions(n):
    try:
        with psycopg2.connect(dbname=DATABASE_NAME, user=USER, password=PASSWORD, host=HOST) as conn:
            with conn.cursor() as cur:
                cur.execute(sql.SQL("""
                    INSERT INTO historical_transactions (tx_hash, gas_used, timestamp)
                    SELECT tx_hash, gas_used, timestamp FROM current_transactions
                    ORDER BY timestamp DESC LIMIT %s
                """), (n,))
    except (Exception, DatabaseError) as e:
        print(f"Database error occurred: {e}")

# Function to insert anomaly data
def insert_anomaly(tx_hash, gas_used):
    try:
        with psycopg2.connect(dbname=DATABASE_NAME, user=USER, password=PASSWORD, host=HOST) as conn:
            with conn.cursor() as cur:
                cur.execute(sql.SQL("""
                    INSERT INTO anomalies (tx_hash, gas_used)
                    VALUES (%s, %s)
                """), (tx_hash, gas_used))
    except (Exception, DatabaseError) as e:
        print(f"Database error occurred: {e}")

# Function to get anomaly history
def get_anomaly_history():
    try:
        with psycopg2.connect(dbname=DATABASE_NAME, user=USER, password=PASSWORD, host=HOST) as conn:
            with conn.cursor() as cur:
                cur.execute(sql.SQL('SELECT * FROM anomalies'))
                anomalies = cur.fetchall()
                return anomalies
    except (Exception, DatabaseError) as e:
        print(f"Database error occurred: {e}")
        return []

# Function to get the previous gas usage
def get_historical_gas_usage():
    try:
        with psycopg2.connect(dbname=DATABASE_NAME, user=USER, password=PASSWORD, host=HOST) as conn:
            with conn.cursor() as cur:
                cur.execute(sql.SQL('SELECT * FROM historical_transactions'))
                historical_transactions = cur.fetchall()
                return historical_transactions
    except (Exception, DatabaseError) as e:
        print(f"Database error occurred: {e}")
        return []

# Function to get current gas usage
def get_gas_usage_data():
    try:
        with psycopg2.connect(dbname=DATABASE_NAME, user=USER, password=PASSWORD, host=HOST) as conn:
            with conn.cursor() as cur:
                cur.execute(sql.SQL('SELECT * FROM current_transactions'))
                current_transactions = cur.fetchall()
                return current_transactions
    except (Exception, DatabaseError) as e:
        print(f"Database error occurred: {e}")
        return []

# Main
if __name__ == "__main__":
    create_current_transactions_table()
    create_historical_transactions_table()

```

The provided Python script facilitates data management for gas usage information from transactions in a PostgreSQL database named `gas_usage`.

Components:

Modules Imported: `psycopg2` and `sql` from `psycopg2` utilized for PostgreSQL database interaction.

Database Credentials:

`DATABASE_NAME`, `USER`, `PASSWORD`, and `HOST` hold respective database connection details.

Connection Establishment:

Connection (`conn`) and cursor (`cur`) objects are created to interact with the PostgreSQL database.

Functionality:

Table Creation Functions:

`create_current_transactions_table()`: Establishes the `current_transactions` table if not present, storing `tx_hash`, `gas_used`, and `timestamp`.

`create_historical_transactions_table()`: Creates the `historical_transactions` table for storing similar transaction data.

Data Manipulation Functions:

`insert_current_transaction(tx_hash, gas_used, timestamp)`: Inserts a transaction record into `current_transactions`.

`store_historical_transactions(n)`: Transfers the last `n` transactions from `current_transactions` to `historical_transactions`.

`insert_anomaly(tx_hash, gas_used)`: Inserts anomaly data into a table named `anomalies` (this table definition is missing in the provided code).

`get_anomaly_history()`: Retrieves historical anomaly data.

`get_historical_gas_usage()`: Fetches historical gas usage transaction data.

`get_gas_usage_data()`: Retrieves current gas usage transaction data.

Table Structures:

`current_transactions` and `historical_transactions` tables:

Defined with columns: `id` (SERIAL PRIMARY KEY), `tx_hash` (TEXT NOT NULL), `gas_used` (INTEGER NOT NULL), and `timestamp` (TIMESTAMP NOT NULL).

`id` serves as a unique identifier, while other columns store relevant transaction information.

Operations:

Table Creation:

The script executes functions to create essential tables for managing current and historical transaction data.

Usage Example:

An example of inserting current transaction data and storing historical transactions is provided (commented out).

Proper closure of the database cursor (cur) and connection (conn) ensures resource release and data integrity.

The **initial SQLite** code lacked crucial elements such as error handling, centralized connection management, and sophisticated CRUD operations seen in the PostgreSQL code. Modifications were made to the SQLite code to mirror the PostgreSQL structure, incorporating enhanced connectivity management, explicit error handling, and more granular CRUD functionalities. The PostgreSQL code demonstrated a clear separation between current and historical transaction tables, with specific functions for insertion, retrieval, and anomaly handling, promoting a more modular and efficient approach.

Phase 2: Data Collection Script [2]

```
import time
import os
from web3 import Web3, HTTPProvider
import Database
import app

# API Key and contract details
INFURA_PROJECT_ID = os.getenv('Brownie-Detector')
CONTRACT_ADDRESS = os.getenv(app.form_data['blockchain'])

# Connect to the Ethereum Infrastructure
infura_url = f'https://mainnet.infura.io/v3/{INFURA_PROJECT_ID}'
web3 = Web3(HTTPProvider(infura_url))
contract_address = Web3.toChecksumAddress(CONTRACT_ADDRESS)

# Function to collect gas usage data
def collect_gas_usage():
    if not web3.isConnected():
        print("Failed to connect to Infura. Retrying...")
        time.sleep(10)
        return

    try:
        block = web3.eth.get_block('latest', full_transactions=True)
        for tx in block.transactions:
            if tx.to and Web3.toChecksumAddress(tx.to) == contract_address:
                tx_receipt = web3.eth.getTransactionReceipt(tx.hash)
                # Insert the transaction data into the current_transactions table
                Database.insert_current_transaction(tx.hash.hex(), tx_receipt.gasUsed, block.timestamp)
                break

        total_transactions = len(block.transactions)
        n = app.form_data['num_transactions'] - 1

        if n > total_transactions:
            n = total_transactions

        # Insert rest of the transaction in the historical transactions table
        for i in range(1, n):
            Database.store_historical_transactions(block.transactions[-i])

    except Exception as e:
        print(f"An error occurred: {e}")
    finally:
        time.sleep(10) # Limit key requests

if __name__ == "__main__":
    collect_gas_usage()
```

For data_gathering.py comparison : the modified code, which utilizes the Database module, performs similar data collection but with enhancements. It includes more comprehensive error handling and a structured approach to database interaction. This revised version explicitly differentiates between current and historical transactions, utilizing distinct tables in the database. The function collect_gas_usage() retrieves the latest block transactions and inserts the relevant data into the current_transactions table. It then iterates through previous transactions, storing them in the historical_transactions table for historical record-keeping. This improved version provides better error handling, more structured database interaction, and separates historical data for improved data management and retrieval. The addition of a conditional check before inserting historical transactions ensures data integrity and reduces redundancy, enhancing the efficiency of the data collection process.

The code we provided in iteration2 is **static**, it doesn't take any user interaction but the code we implemented is more of **dynamic**.

Phase 3: Anomaly Detection Script

```
import numpy as np
import Database
from anomaly_alert import send_email_alert

def fetch_gas_data():
    return Database.get_gas_usage_data()

def fetch_historical_gas_data():
    return Database.get_historical_gas_usage()

def compute_std_dev_threshold(historical_gas_data):
    gas_values = [x[0] for x in historical_gas_data] # Assuming x[0] is the gas_used value
    mean_gas = np.mean(gas_values)
    std_dev_gas = np.std(gas_values)
    return mean_gas, std_dev_gas

def is_anomaly(current_gas, mean_gas, std_dev_gas):
    return abs(current_gas - mean_gas) > std_dev_gas

def compute_anomalies(current_gas_data, mean_gas, std_dev_gas):
    return [data for data in current_gas_data if is_anomaly(data['gas_used'], mean_gas, std_dev_gas)]

def main():
    historical_gas_data = fetch_historical_gas_data()
    mean_gas, std_dev_gas = compute_std_dev_threshold(historical_gas_data)

    current_gas_data = fetch_gas_data()
    anomalies = compute_anomalies(current_gas_data, mean_gas, std_dev_gas)

    for anomaly in anomalies:
        print(f"Anomaly detected for transaction {anomaly['tx_hash']} with {anomaly['gas_used']} gas used.")
        alert_message = f"Anomaly detected for transaction {anomaly['tx_hash']} with {anomaly['gas_used']} gas used."
        send_email_alert("Anomaly Detected", alert_message, "mittalaman026@gmail.com")
        Database.insert_anomaly(anomaly['tx_hash'], anomaly['gas_used'])

if __name__ == "__main__":
    main()
```

Anomaly detection introduces more structured functionalities. It includes a dedicated function `fetch_historical_gas_data()` to retrieve historical gas data separately from current data and a distinct `compute_std_dev_threshold()` function to calculate the mean and standard deviation specifically for historical data. This separation ensures more accurate anomaly detection by employing statistics from historical records. The `is_anomaly()` function improves readability by encapsulating the anomaly detection logic. Furthermore, it incorporates the `Database.insert_anomaly()` function to store detected anomalies in the database, enabling historical tracking.

Phase 4: Analysis

Initializing the Database

Need to initialize the db before running the code:

```
flask db init
flask db migrate -m "Initial migration."
flask db upgrade
```

When setting up a new application that uses Flask with a database, we often need to perform initial setup operations to prepare our database to work with the application. This setup typically involves creating the necessary database tables and structures that.

1. flask db init:

This command sets up the migration environment. It creates the `migrations` folder in which will contain all the migration scripts. This is necessary for tracking changes to the database schema. Running this command is a one-time setup operation at the beginning of the project.

2. flask db migrate -m "Initial migration:"

This command creates a new migration script. It does so by detecting changes to the database schema based on the models defined in the Flask application. The `-m` option allows us to add a human-readable message to the migration script, in this case, "Initial migration."

3. flask db upgrade:

This command applies the migration script to the actual database. It will run the `upgrade()` method defined in the migration script, which typically includes commands to alter the database schema, such as creating new tables or modifying existing ones. After this command, the database should have all the tables and columns defined by your Flask models, and it will be ready for use by your application.

Anomaly Alert

```
import os
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
import app

def send_email_alert(subject, message, recipient_email):
    sender_email = os.getenv('SYSTEM_EMAIL')
    sender_password = os.getenv('SYSTEM_EMAIL_APP_PASSWORD')

    msg = MIMEMultipart()
    msg['From'] = sender_email
    msg['To'] = app.form_data["email"]
    msg['Subject'] = subject
    msg.attach(MIMEText(message, 'plain'))

    server = smtplib.SMTP('smtp.gmail.com', 587)
    server.starttls()
    server.login(sender_email, sender_password)
    server.send_message(msg)
    server.quit()
```

The Python script `anomaly_alert.py` appears to be designed for sending email alerts. Here's a brief overview of its functionality.

1. The script imports modules like `os` for interacting with the operating system, `smtplib` for handling SMTP (Simple Mail Transfer Protocol) which is used for sending emails, and `email.mime` components for creating email messages.

2. The core functionality of the script is encapsulated in a function named `send_email_alert`. This function is designed to send an email with a specified subject and message to a given recipient's email address.

3. Email Configuration and Sending:

The function retrieves the sender's email address and password from the environment variables `SYSTEM_EMAIL` and `SYSTEM_EMAIL_APP_PASSWORD`. This implies that the script is expected to be run in an environment where these variables are set for security reasons.

It constructs an email message using `MIMEMultipart`, setting appropriate headers for 'From', 'To', and 'Subject'.

The email's body is added as plain text.

The script sets up a connection to Gmail's SMTP server (`smtp.gmail.com`) using port 587, which is standard for email sending.

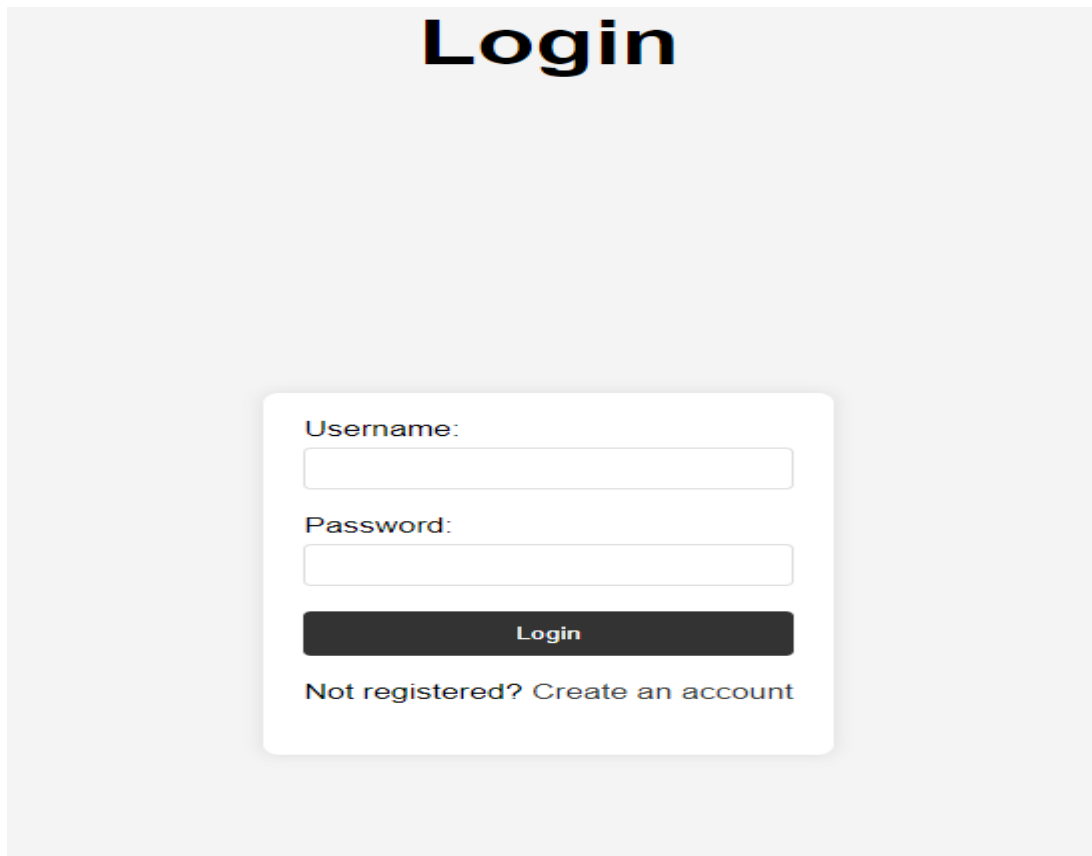
4. SMTP Server Interaction

The script starts to establish a connection with the SMTP server using the `smtplib.SMTP` object.

Basically, the script is focused on automating the process of sending email notifications, likely for alerting purposes, such as notifying administrators or users about anomalies or important events in a system or application.

After that we have implemented the frontend for the website using html and css.

Login



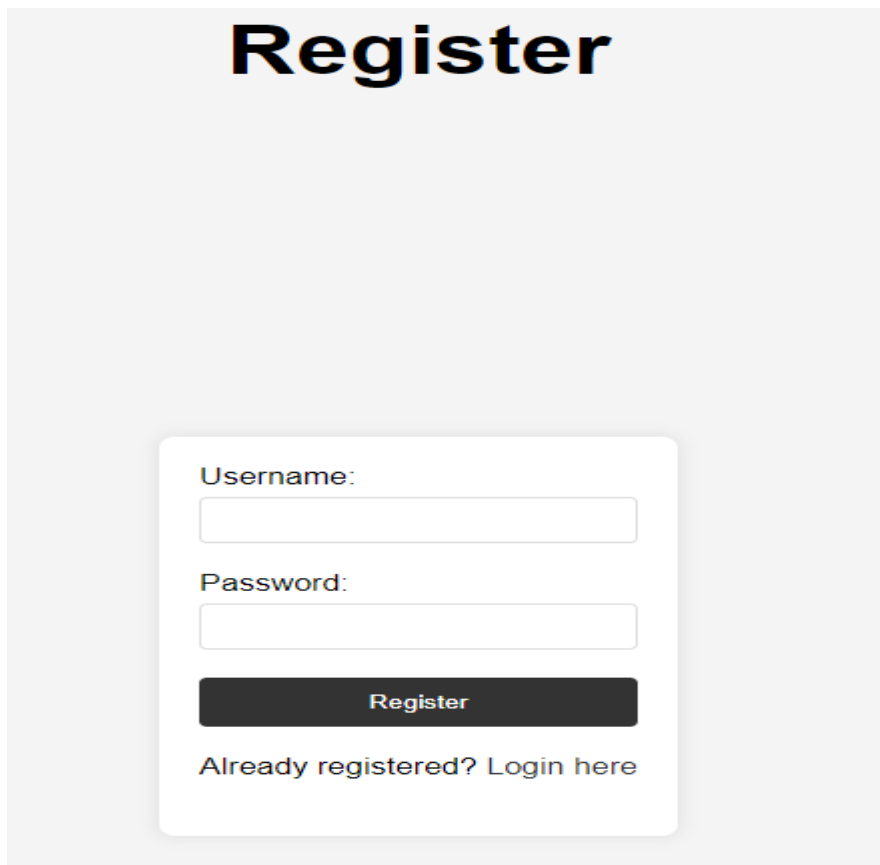
The image shows a login page with a light gray background. At the top center, the word "Login" is written in a large, bold, black font. Below it, centered, is a white rectangular form with rounded corners and a subtle drop shadow. Inside the form, the label "Username:" is followed by a text input field. Below that, the label "Password:" is followed by another text input field. Under the password field is a dark gray button with the word "Login" in white text. At the bottom of the form, the text "Not registered? Create an account" is displayed, with "Create an account" being a clickable link.

The login page is designed to authenticate users. It is a gateway for users to access their accounts and interact with the application.

It references an external stylesheet, `style.css`, suggesting a consistent design across the application.

The form action uses a dynamic routing mechanism (`{{ url_for('login') }}`), indicating integration with a backend framework, likely Flask.

Register Page:

The image shows a registration form on a light gray background. At the top, the word "Register" is written in a large, bold, black font. Below it, there is a white rounded rectangle containing the form fields. The first field is labeled "Username:" and is an empty text input. The second field is labeled "Password:" and is also an empty text input. Below these fields is a dark gray button with the word "Register" in white. At the bottom of the white box, there is a link that says "Already registered? Login here".

Register

Username:

Password:

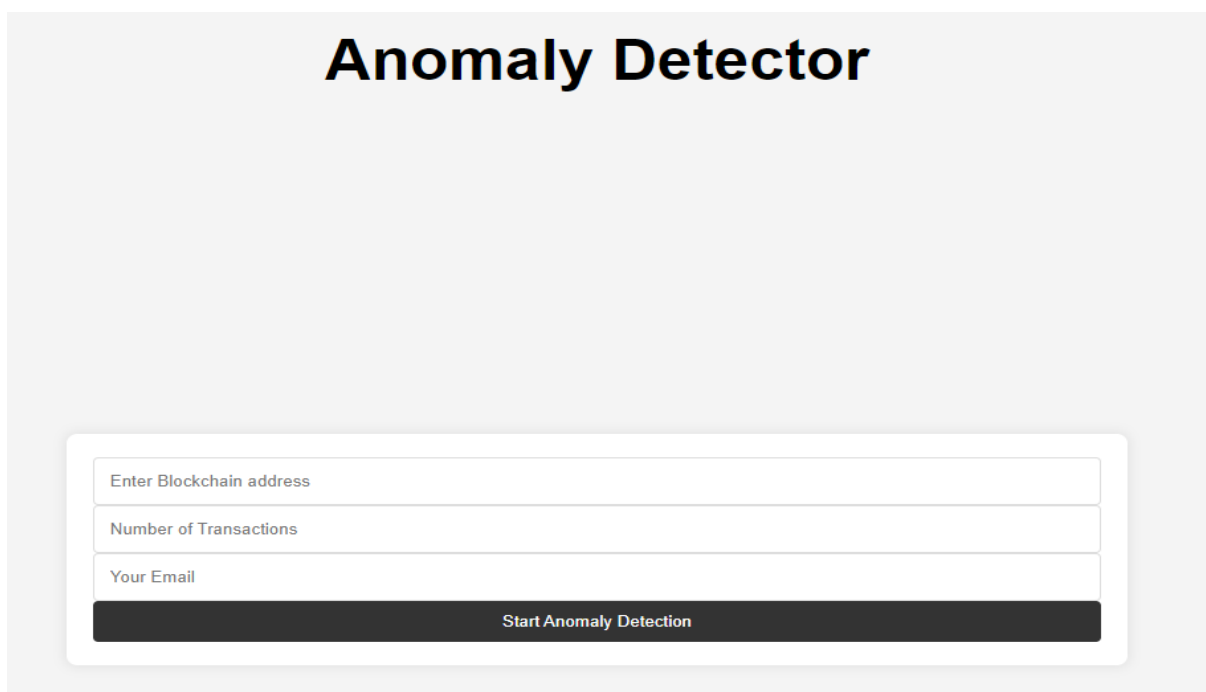
Register

Already registered? [Login here](#)

The registration page is crucial for onboarding new users, allowing them to create new accounts.

The form submission is handled by a server-side route (`{{ url_for('register') }}`), which likely processes the registration data.

Detector Anomaly part

The image shows an "Anomaly Detector" form on a light gray background. At the top, the words "Anomaly Detector" are written in a large, bold, black font. Below it, there is a white rounded rectangle containing the form fields. The first field is labeled "Enter Blockchain address". The second field is labeled "Number of Transactions". The third field is labeled "Your Email". Below these fields is a dark gray button with the text "Start Anomaly Detection".

Anomaly Detector

Enter Blockchain address

Number of Transactions

Your Email

Start Anomaly Detection

This interface is tailored for the application's core functionality, which involves submitting data for anomaly detection in blockchain transactions.

The analyzed HTML pages are integral components of a blockchain-related web application. The login and registration pages facilitate secure user access, while the anomaly detection form ties directly to the application's primary purpose. The consistent use of an external stylesheet across pages ensures a unified user experience. The backend integration, as indicated by the form actions, suggests that these interfaces are part of a dynamic and interactive web application, possibly built using Python and Flask. The anomaly detection form, in particular, stands out as a specialized tool within this application, emphasizing its focus on blockchain technology.

```
body {
  font-family: Arial, sans-serif;
  display: flex;
  justify-content: center;
  align-items: center;
  height: 100vh;
  background-color: #f4f4f4;
}

form {
  background: #fff;
  padding: 20px;
  border-radius: 8px;
  box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
}

h2 {
  font-size: 3em; /* Increase font size */
  position: absolute; /* Position the element */
  top: 0; /* Align to the top of the viewport */
  left: 50%; /* Center horizontally (optional) */
  transform: translateX(-50%);
}

div {
  margin-bottom: 20px;
}

label {
  display: block;
  margin-bottom: 5px;
}

input[type="text"],
input[type="password"] {
  width: 100%;
  padding: 8px;
  border: 1px solid #ddd;
  border-radius: 4px;
  box-sizing: border-box;
}

button {
  width: 100%;
  padding: 10px;
  border: none;
  border-radius: 4px;
  background-color: #333;
}
```

This is the styling script we have used for the above pages.

```

from flask import Flask, render_template, request, redirect, url_for, session, flash
from flask_sqlalchemy import SQLAlchemy
from flask_migrate import Migrate
from werkzeug.security import generate_password_hash, check_password_hash

app = Flask(__name__)
app.secret_key = 'oiahre3ijrsdmvlskhfskdncvpaokfanfaslmclkanfjsdnflskdjvisl'

# PostgreSQL database configuration
username = 'Aman_Mittal'
password = 'Password@123'
host = 'localhost'
dbname = 'registered_users'
app.config['SQLALCHEMY_DATABASE_URI'] = f'postgresql://{username}:{password}@{host}/{dbname}'

db = SQLAlchemy(app)
migrate = Migrate(app, db)

class User(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    username = db.Column(db.String(50), unique=True, nullable=False)
    password_hash = db.Column(db.String(80))

    def set_password(self, password):
        self.password_hash = generate_password_hash(password)

    def check_password(self, password):
        return check_password_hash(self.password_hash, password)

@app.route('/register', methods=['POST'])
def register():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        user = User(username=username)
        user.set_password(password)
        db.session.add(user)
        db.session.commit()

        flash('Registered successfully! Please login.')
        return redirect(url_for('login'))

    return render_template('register.html')

```

This is the explanation for the backend component of a web application, focusing on the `app.py` file.

The `app.py` file serves as the central part of the web application's backend, handling configurations, database interactions, and routing.

```

@app.route('/login', methods=['POST'])
def login():
    if request.method == 'POST':
        username = request.form['username']
        password = request.form['password']

        user = User.query.filter_by(username=username).first()
        if user and user.check_password(password):
            session['loggedin'] = True
            session['username'] = username
            return redirect(url_for('index'))

        flash('Invalid username or password')
    return render_template('login.html')

form_data = {}

@app.route('/detect', methods=['POST'])
def detect():
    global form_data
    form_data['blockchain'] = request.form['blockchain']
    form_data['num_transactions'] = int(request.form['num_transactions'])
    form_data['email'] = request.form['email']

    # Call a function from data_gathering.py or handle data processing here
    # For example: data_gathering.process_data(form_data)

    return 'Anomaly Detection Started'

```

The script imports Flask, which is a micro web framework written in Python, used for developing web applications.

- render_template: To render HTML templates.
- request, redirect, url_for, session, flash: These are used for handling HTTP requests, redirections, URL building, session management, and flashing messages.
- SQLAlchemy: An Object-Relational Mapping (ORM) library for Python, used for database interactions.
- Migrate: Flask-Migrate is an extension that handles SQLAlchemy database migrations.
- app = Flask(__name__)` initializes the Flask application.
- Secret Key: A secret key is set for sessions and other security features.
- Database Configuration: The script includes configuration details for a PostgreSQL database, such as username, password, host, and database name.
- User Authentication: Given the import of password hashing and session management utilities, the application likely handles user authentication processes.
- Database Operations: The use of SQLAlchemy and Migrate suggests that the application performs operations like storing, retrieving, and managing data in a PostgreSQL database.

- Routing and Web Page Rendering: With Flask, the script is expected to define various routes (URLs) that handle different parts of the web application, such as user registration, login, and perhaps anomaly detection, as inferred from the previously analyzed HTML files.

In conclusion, `app.py` is configured to serve as the backbone of a dynamic web application, likely providing user authentication, database interaction, and web page rendering functionalities. The use of Flask alongside extensions like SQLAlchemy and Migrate indicates a robust, database-driven architecture, suitable for handling user data and application-specific operations such as anomaly detection in a blockchain context. The script's initial setup lays a foundation for a secure, efficient, and scalable web application.

1. Feature Implementation Plan

Features that provide the most value to users and have the least technical complexity should be implemented first. This approach minimizes risk and ensures early delivery of valuable functionalities.

Phase 1: Core Features

- Login/Registration System: Essential for user management and security.
- Basic Anomaly Detection: A simplified version focusing on the most critical aspects of gas usage monitoring.
- User Feedback System: To gather early user insights and improve iteratively.

Phase 2: Advanced Features

- Enhanced Anomaly Detection Algorithms: Incorporating more sophisticated algorithms based on initial user feedback and data.
- Detailed Reporting Tools: Providing users with in-depth analysis of gas usage.
- Customizable Alerts: Allowing users to set thresholds and preferences for alerts.

Phase 3: Additional Features

- Integration with External Blockchain Services: To broaden the scope of monitoring.
- User Dashboard Enhancements: Introducing more interactive and detailed visualizations.

2. Overview of Related Apps

Competitive Landscape:

- Identify apps that provide blockchain monitoring and anomaly detection.
- Compare features, user interfaces, and performance metrics.

3. Differentiation Strategy

Unique Selling Propositions (USPs):

- Specialized Focus on Gas Usage: Unlike general blockchain monitoring tools.
- User-Centric Design: Prioritizing ease of use and actionable insights.

- Advanced Analytics: Providing deeper insights than competitors.

4. Top Five Risks

1. Technological Complexity: Risk of underestimating the complexity of blockchain analytics.
2. User Adoption: Difficulty in convincing users to switch from existing solutions.
3. Data Security and Privacy: Handling sensitive blockchain data securely.
4. Regulatory Compliance: Navigating the evolving landscape of blockchain regulation.
5. Scalability: Ensuring the system scales with user growth and blockchain evolution.

Operational Risks: Inaccuracies in data collection or analysis could lead to false alerts or missed detections.

Security Risks: As a tool dealing with blockchain data, it's a target for cyberattacks, risking data integrity and confidentiality.

Technological Risks: Rapid changes in blockchain technology and Ethereum updates may require constant tool updates to stay relevant.

Compliance Risks: The tool must comply with evolving data privacy laws and blockchain regulations.

5. Risk Mitigation Plans

- Technological Complexity: Invest in expert consultations and allocate time for R&D.
- User Adoption: Engage with the community early, and offer incentives for trial usage.
- Data Security and Privacy: Implement robust security protocols and conduct regular audits.
- Regulatory Compliance: Stay updated with regulations and adapt the system accordingly.
- Scalability: Design with scalability in mind, using cloud services and modular architecture.

Inputs:

1. User Credentials: For any user login functionality.
2. Gas Usage Data: Typically includes timestamps, gas amounts, user or sensor identifiers, etc.
3. Configuration Settings: Thresholds for anomalies, alert configurations, etc.
4. User Feedback: On reported anomalies for refining detection algorithms.

Outputs:

1. Anomaly Alerts: Notifications sent to users or systems when an anomaly is detected.
2. Reports: Detailed reports of gas usage and detected anomalies.
3. Statistical Data: Summaries, averages, trends of gas usage over time.
4. System Logs: Records of system events, errors, and user actions.

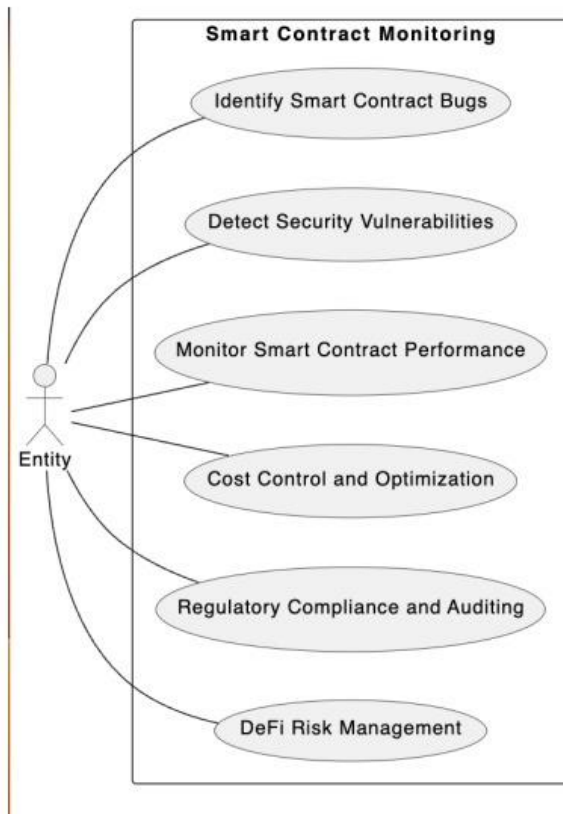
Key Data Structures:

1. User Table: Stores user credentials, roles, and profile information.
2. Gas Usage Records: Stores individual records of gas usage data points.
3. Anomaly Records: Stores details of detected anomalies.
4. Configuration Table: Stores system and user-specific configuration settings.

Use Cases:

Here are several key use cases -

1. Identifying Smart Contract Bugs: Sometimes a bug in a smart contract may cause functions to consume more gas than they should. An anomaly detector can flag such unusual gas usage patterns for further investigation.
2. Detecting Security Vulnerabilities: Certain types of security breaches, such as reentrancy attacks or other exploits, can result in abnormal gas usage. Early detection of such anomalies can prevent further exploitation.
3. Monitoring Smart Contract Performance: If a contract suddenly starts using more gas for typical operations, it may be due to an inefficient code update or change in the contract's state, signaling the need for optimization.
4. Cost Control and Optimization: For entities that interact with smart contracts frequently, monitoring for anomalies in gas usage can help manage and reduce transaction costs.
5. Regulatory Compliance and Auditing: For regulated entities, monitoring gas usage can be part of compliance with regulations that require monitoring and reporting of transactional activity.
6. Decentralized Finance (DeFi) Risk Management: In DeFi, where contracts handle various financial operations, unusual gas patterns might indicate market manipulation or failing contract mechanisms, which can be crucial for risk management.



Exceptional Case Coverage:

1. Authentication Failure: What occurs when a user fails to log in.
2. Data Ingestion Errors: How the system handles incorrect or malformed input data.
3. No Anomaly Found: System behavior when no anomaly is detected in the data.
4. System Failures: Backup and recovery processes in case of system crashes or errors.

The project focuses on developing an anomaly detection system for gas usage in smart contracts on the Ethereum blockchain.

1. Customers and Users Identification

The primary objective of the Gas Usage Anomaly Detector is to provide early warnings of irregular patterns in gas usage, which could indicate inefficiencies or security concerns.

Identified Customer Segments:

1. Blockchain Developers and Companies:

- Needs: Seeking tools to optimize smart contract efficiency and ensure security against potential threats.
- Feedback: Interested in detailed analytics of gas usage and regular updates on anomaly detection algorithms.

2. Smart Contract Auditors:

- Needs: Require tools for auditing smart contracts for potential inefficiencies and vulnerabilities.
- Feedback: Expect precise and reliable reports on gas usage anomalies.

3. Blockchain Enthusiasts and Researchers:

- Needs: Looking for insights into the efficiency and security aspects of smart contracts.
- Feedback: Interested in comprehensive data and case studies on detected anomalies.

2. Feedback Mechanisms and User Interaction

Positive Aspects:

Users appreciate The tool's ability to provide real-time monitoring of gas usage enabled prompt detection and resolution of irregularities. This not only streamlined their operations but also led to significant cost savings.

Areas for Improvement:

Users mentioned the need for more detailed analytics and customizable alert thresholds.

Feedback suggests a demand for better user interfaces and easier integration with existing development tools.

Overall Reception:

Generally positive reception, especially from users who prioritize smart contract efficiency and security

Strategies for Frequent Feedback:

1. Surveys and Feedback Forms: Regularly distributed surveys to gather feedback on the tool's effectiveness and areas for improvement.
2. Community Forums: Online forums for users to discuss their experiences, share insights, and provide suggestions.
3. Customer Support Channels: Dedicated channels for technical support and user queries.

Documentation of User Interactions:

1. Case Studies: Documenting specific instances where the anomaly detector successfully identified issues, including user testimonials.
2. Usage Reports: Periodic reports summarizing user interactions, feedback trends, and system performance metrics.
3. User Experience Studies: Conducting studies to understand how different user segments interact with the system and their satisfaction levels.

The project benefits from a proactive approach to user feedback and interaction documentation. This includes regular surveys, community engagement, and detailed case studies. his customer-centric approach will ensure the project remains relevant, effective, and ahead of emerging trends in blockchain technology.

CATEGORY	Tools and Technologies
Web3 Libraries	Web3.js , Ether.js
Data Storage	PostgreSQL
Backend Framework	Flask
Front-end Framework	HTML , CSS
APIs for Alerting	Email services , Communication Platforms

Conclusion :

The system implements an anomaly detection model to identify unusual gas usage patterns in transactions involving the specified contract.

It calculates the standard deviation and mean of gas used in historical transactions and flags any current transactions that deviate significantly from these statistics as anomalies.

We're developing a web application using the Flask framework that allows users to register and log in to access a specialized service.

The application uses 'web3.py' library to connect to the Ethereum blockchain via Infura, allowing it to read block and transaction data in real-time.

The application leverages PostgreSQL to store transaction data in Different tables.

Upon detecting an anomaly, the system triggers an alert mechanism.

References:

1. <https://docs.python.org/3/library/sqlite3.html>
2. <https://moralis.io/python-and-web3-a-web3-and-python-tutorial-for-blockchain-development/>
3. <https://www.postgresql.org/docs/current/tutorial-createdb.html>
4. <https://www.prisma.io/dataguide/postgresql/setting-up-a-local-postgresql-database>
5. <https://www.postgresql.org/docs/current/ecpg-errors.html>
6. <https://www.dappuniversity.com/articles/web3-js-intro>
7. <https://ethereum.stackexchange.com/questions/116945/how-to-get-full-info-of-a-transaction-hash-web3-py-or-web3-js>
8. <https://www.geeksforgeeks.org/numpy-std-in-python/>
9. <https://www.w3schools.com/css/>
10. <https://flask.palletsprojects.com/en/3.0.x/>
11. Flask Tutorial: <https://www.youtube.com/watch?v=MwZwr5Tvyxo&list=PL-osiE80TeTs4UjLw5MM6OjgkjFeUxCYH>