ITERATION 2

THE UNIVERSITY OF TEXAS AT ARLINGTON

Advanced Topics in Software Engineering CSE 6324 – Section 001

Team 07

ITERATION02

(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

BROWNIE :

Part 1: ERRORS WITH VYPER TESTS [1]

Technical Issue Report: Type Conversion Failure in Vyper Smart Contracts Testing:

Vyper: An alternative language to Solidity for writing smart contracts for the Ethereum blockchain. It aims to be more simple and secure with syntax similar to Python.

Ganache: A local blockchain simulator used for Ethereum software development. It allows developers to deploy contracts, develop applications, and run tests.

This report addresses a specific technical issue encountered while testing smart contracts written in Vyper using the Brownie framework. The central problem revolves around a type conversion failure, specifically with the 'int()' function in Python when a base is specified, causing a 'TypeError'. We'll explore the issue, its root cause, and potential solutions.

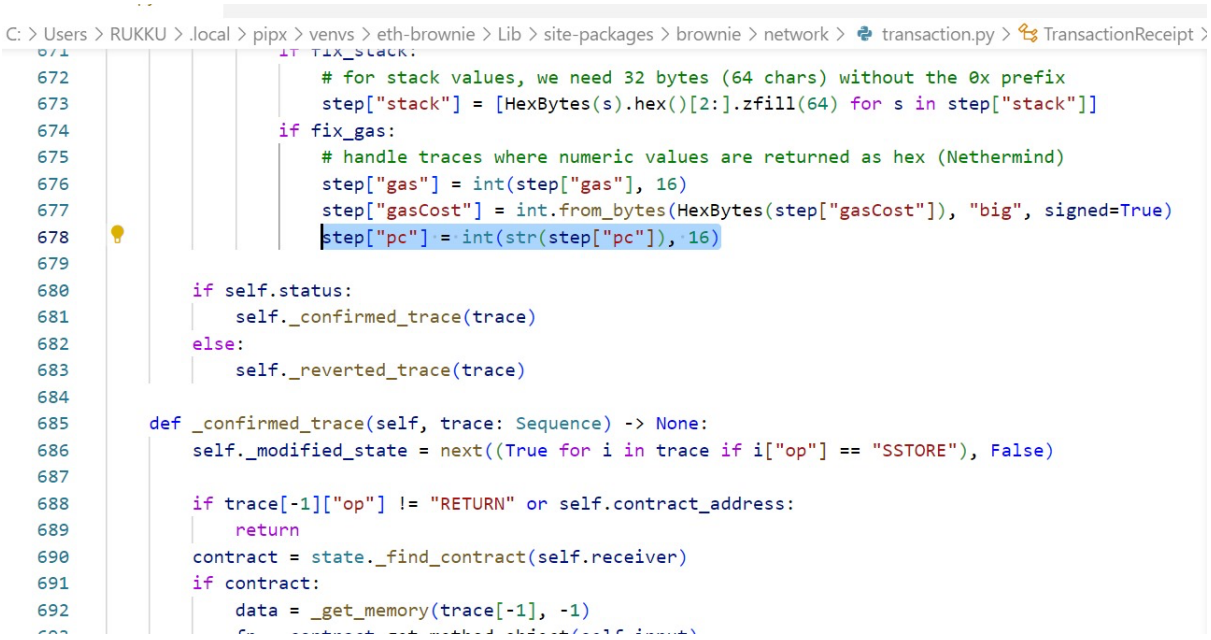
Issue Details

Error Message:

The error message reported is as follows:

> **TypeError**: 'int() can't convert non-string with explicit base'
>
> The Python int() function threw a TypeError because it was expecting a string input when a base was specified, but it received something else. You can clarify that specifying a base is common when dealing with numerical representations like hexadecimal numbers.

Line caused error:



Mismatched Argument Type

The primary issue in the code as above mentioned:

```
step["pc"] = int(step["pc"], 16)
```

In Python, int(x, base) will convert x to an integer using the given base. If x is not a string, it must be a number, and the base must be omitted. The error suggests that step["pc"] is not a string when it needs to be. For example, int(1A, 16) would throw an error because '1A' is not in quotes to signify a string.

Solution:

Implemented solution:

Built-in Formatting Function: Using format () is incorrect for type conversion. It should be used to format an integer as a hexadecimal string. This method is inappropriate for type conversion, as it can cause errors if 'step["pc"]' is not already an integer.

```
step["pc"] = int(format(step["pc"], 'x'), 16)
```

after implementation we found it works for particular versions but again few versions are giving trouble. But following solution which has been commented on GitHub works to solves this bug.

Suggested solution:

Explicit String Conversion [1]: This approach ensures step["pc"] is a string before converting it to an integer with a hexadecimal base, regardless of its original type. However, it assumes that step["pc"] has a meaningful string representation.

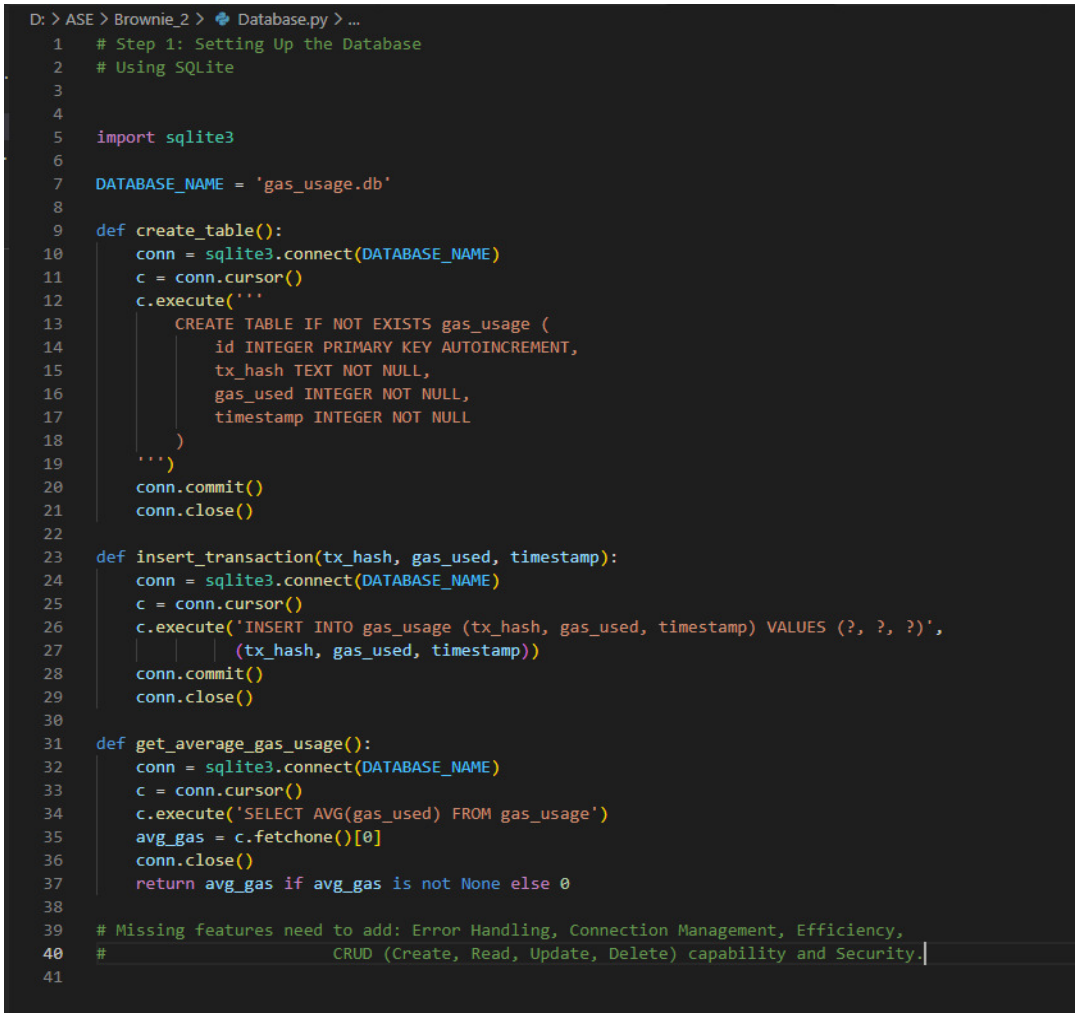
```
step["pc"] = int(str(step["pc"]), 16)
```

Part 2: GAS USAGE AN0MALY 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.

Phase 1: Set Up the Database [9]



The provided Python script sets up a SQLite database named `gas_usage.db` to store gas usage data from Ethereum transactions. The `create_table` function establishes a `gas_usage` table with columns for `id`, `tx_hash`, `gas_used`, and `timestamp`. This step has been completed, and the database is ready to store the collected data.

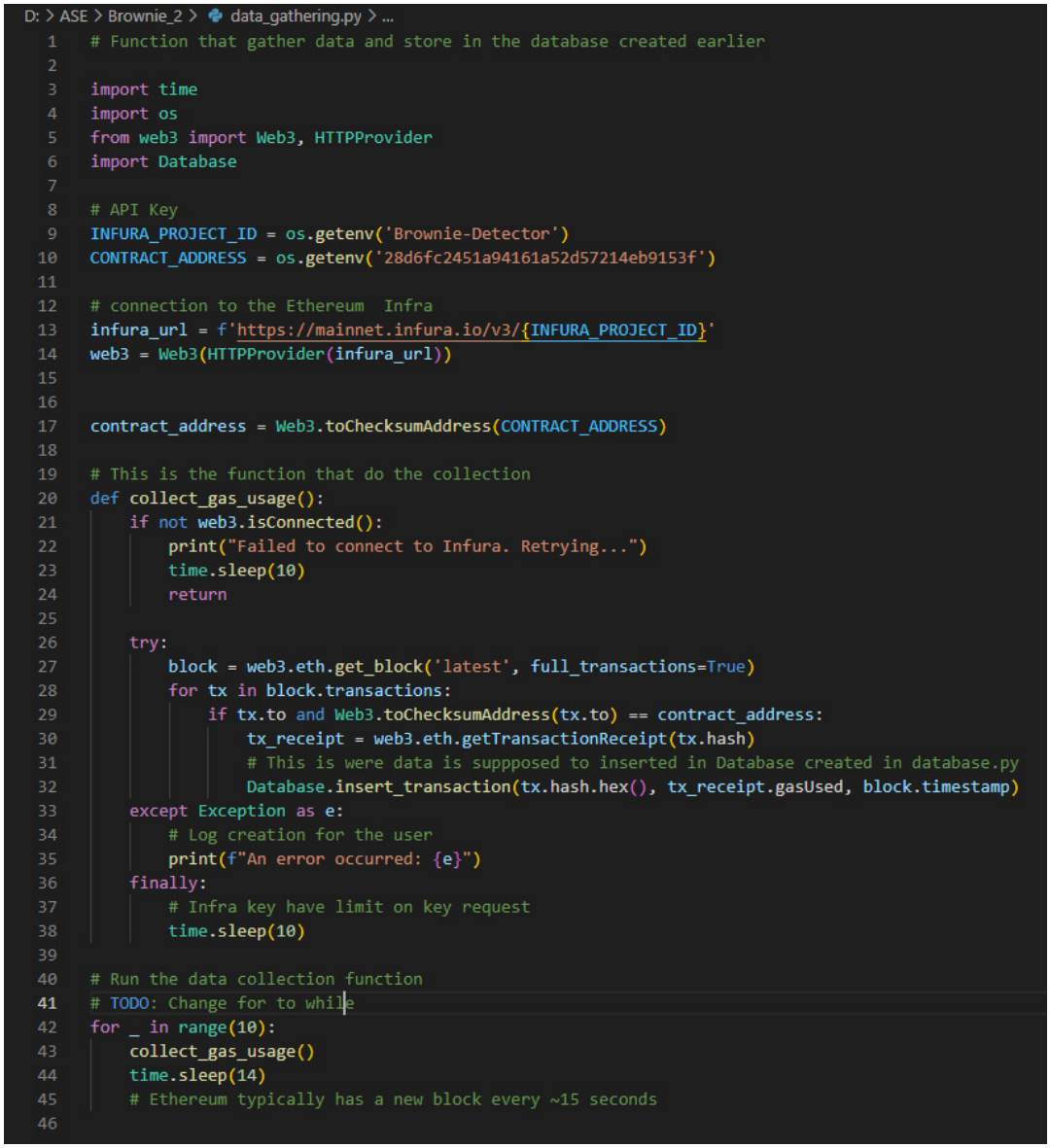
Here's a breakdown of the script:

- **import sqlite3**: This imports the **sqlite3** module, which is a built-in Python library for working with SQLite databases.
- **DATABASE_NAME**: A constant that holds the name of the database file.
- **create_table()**: A function that, when called, will create a connection to the SQLite database and create a table named **gas_usage** if it doesn't already exist.
- The **CREATE TABLE** SQL statement defines the structure of the table with four columns: **id**, **tx_hash**, **gas_used**, and **timestamp**.
    - **id**: A unique identifier for each record, which auto-increments for every new entry.
    - **tx_hash**: A text field that stores the transaction hash. It is marked as **NOT NULL**, meaning it must have a value.

- **gas_used**: An integer field that records the amount of gas used for the transaction.
- **timestamp**: An integer field that records the time at which the transaction occurred.

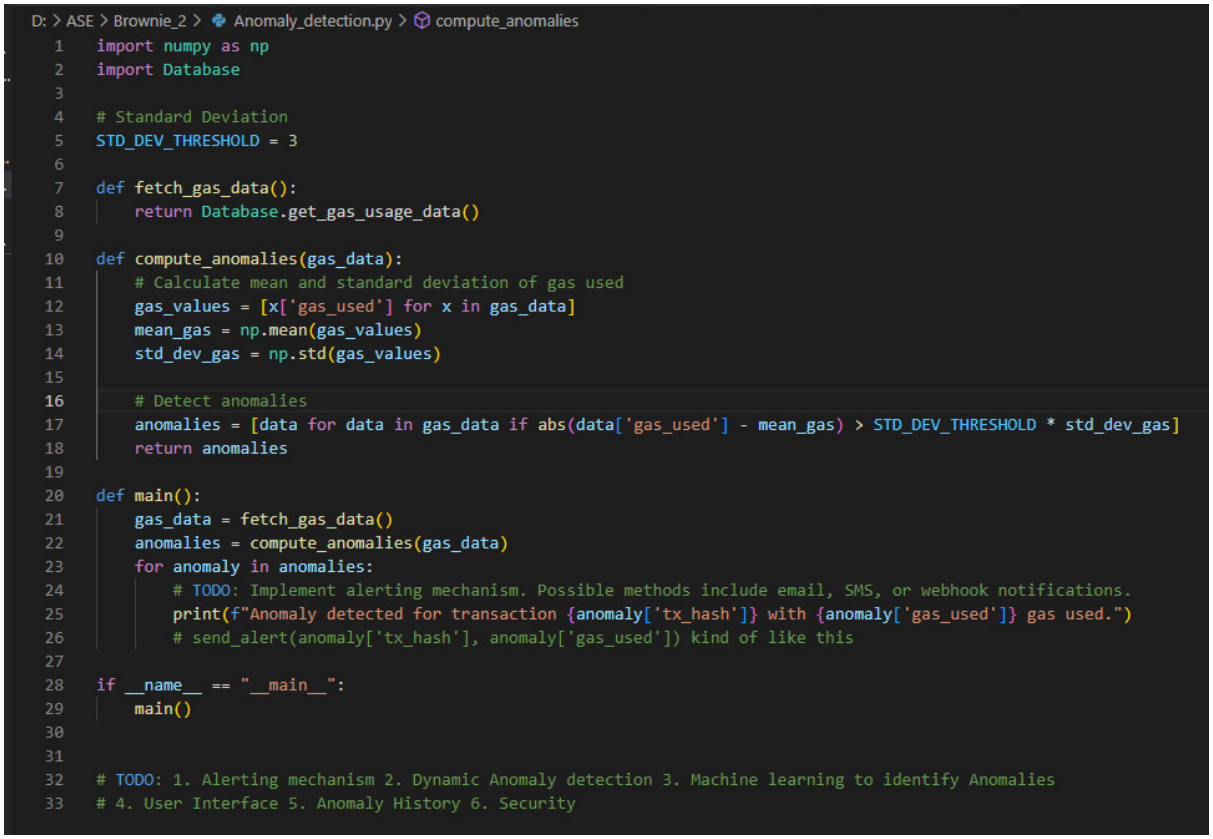
- **conn.commit()**: This saves the changes to the database.
- **conn.close()**: This closes the connection to the database.

Phase 2: Data Collection Script [10]



This script is designed to collect gas usage data from the Ethereum blockchain using the `web3.py` library. The environment variables `INFURA_PROJECT_ID` and `CONTRACT_ADDRESS` are used to connect to the Ethereum network and interact with a specified smart contract. The `collect_gas_usage` function is intended to be implemented to fetch and record the gas used for transactions. As of now, the data collection script has been completed up to this stage.

Phase 3: Anomaly Detection Script



The anomaly detection script is partially completed. It includes:

- Importing the `numpy` library for numerical operations and the `Database` module for database interactions.

- A `STD_DEV_THRESHOLD` constant is set to a random value of 3, which will be used to determine the threshold for detecting anomalies.

- The `fetch_gas_data` function is in place to retrieve gas usage data from the database.

- The `compute_anomalies` function is partially implemented. It will contain the logic to detect anomalies based on the standard deviation threshold. The implementation of the anomaly detection logic is underway and will be refined to accurately identify outliers in the gas usage data.

- It imports the **numpy** library, which is a powerful numerical processing library in Python, and the **Database** module, presumably for fetching data from the database .
- **STD_DEV_THRESHOLD**: This constant is likely used to define the threshold for what is considered an anomaly in terms of standard deviations from the mean.
- **fetch_gas_data()**: A function that retrieves gas usage data from the database.
- **compute_anomalies(gas_data)**: A placeholder function where the logic for detecting anomalies in gas usage will be implemented. This might involve statistical analysis to find outliers in the data.

Phase 4: Main Execution and Alerting [7][8]

The main execution script is under development. The current structure includes:

- A `main` function that will call the `fetch_gas_data` and `compute_anomalies` functions to process the gas usage data and identify anomalies.

- A placeholder for alert implementation is provided. The final version will include functionality to alert users of any detected anomalies. The alerting mechanism is planned to be through email or a similar notification system, ensuring users are promptly informed of any unusual gas usage patterns.

- The creation of an executable file is also in progress. This executable will allow users to run the analysis and view outputs easily without delving into the codebase.

Phase 5: Alerting System Development

- Integration with Notification Services: The alerting system will be integrated with various notification services such as email, Slack, SMS, or push notifications. This ensures that users can receive alerts through their preferred communication channels.

- Alert Customization: Users will be able to customize their alert settings. This includes setting thresholds for notifications and selecting specific transactions or patterns for closer monitoring.

- Implementation of Alert Triggers: Alert triggers will be based on the results from the `compute_anomalies` function. If anomalies exceed the set standard deviation threshold, the system will trigger an alert.

- Testing Alert System: The alert system will undergo thorough testing to ensure its responsiveness and accuracy. Simulated anomalies will be used to confirm that alerts are correctly triggered without false positives or negatives.

- User Interface for Alert Management: A user-friendly interface is planned for development, allowing users to manage alert settings, review past alerts, and oversee ongoing alerts with ease, even for those without a technical background.

Differentiation from Related Applications: Unlike many existing applications, this anomaly detection system is specifically tailored for the Ethereum blockchain, offering:

- Real-time monitoring and alerting specifically for gas usage anomalies.
- Advanced statistical and machine learning algorithms for precise detection.
- A user-centric approach that provides actionable insights for smart contract developers and auditors.

Future Enhancements:

Database Efficiency: Optimize database queries to handle large datasets and potentially implement caching mechanisms.

Data Validation: Include checks to validate the data before processing to ensure integrity and correctness.

Anomaly History Tracking: Maintain a comprehensive log of past anomalies to inform future detection and to understand the temporal patterns.

Security Improvements: Enhance data protection measures and secure communication channels to safeguard sensitive transaction information.

Top Five Risks:

1. Inaccurate Anomaly Detection

- Impact: May erode user trust.

- Likelihood: Moderate.

- Strategy: Engage users early, iterative testing, and algorithm refinement.

2. Data Security

- Impact: Could result in significant data breaches.

- Likelihood: Low to moderate due to stringent security measures.

- Strategy: Implement encryption, access controls, and regular audits.

3. **System Scalability**

- Impact: System may not cope with high transaction volumes.

- Likelihood: Moderate.

- Strategy: Utilize scalable cloud services and design for expansion.

4. **Technology Changes**

- Impact: System may become outdated quickly.

- Likelihood: High due to the fast-paced nature of blockchain technology.

- Strategy: Maintain modular system design and stay updated with community developments.

5. **User Adoption**

- Impact: Low adoption could render the system underutilized.

- Likelihood: Moderate.

- Strategy: Develop a user-friendly interface and provide robust documentation and support.

Risk Mitigation Summary:

1. Inaccurate Anomaly Detection

- Mitigation: Engage users for feedback, perform iterative testing, and regularly update detection algorithms.

2. Data Security

- Mitigation: Implement encryption, enforce access controls, and conduct regular security audits.

3. System Scalability

- Mitigation: Design for scalability from the start, use cloud services, and monitor performance.

4. Technology Changes

- Mitigation: Stay updated with Ethereum developments, ensure system modularity, and engage with the developer community.

5. User Adoption

- Mitigation: Develop a user-friendly interface, provide clear documentation, showcase value, and offer trial periods.

Inputs:

- **Gas Usage Data:** This includes transaction identifiers, the gas used by each transaction, and the timestamp when the transaction occurred.
- **User-Defined Thresholds:** Parameters set by users that define what constitutes an anomaly in the context of gas usage. This may include thresholds for gas used, frequency of transactions, etc.
- **Historical Gas Usage Data:** Past data used to establish baseline patterns and inform the anomaly detection algorithm.

Outputs:

- **Anomaly Alerts:** Notifications or alerts generated when the system detects a gas usage anomaly.
- **Anomaly Reports:** Detailed reports that provide information on detected anomalies, including statistical context and potential impacts.
- **Statistical Summaries:** Aggregated data and statistics that summarize gas usage over time, helping users understand the broader context of detected anomalies.

Key Data Structures:

- **GasUsageRecord:** A record structure that holds data for individual transactions.
    - Fields: transaction_id (INT), gas_used (INT), timestamp (DATETIME)
- **AnomalyRecord:** A structure to log details of detected anomalies.
    - Fields: anomaly_id (INT), transaction_id (INT), anomaly_score (FLOAT), detected_at (DATETIME)
- **UserSettings:** A structure to capture user preferences and settings for the detection algorithm.
    - Fields: user_id (INT), threshold_settings (JSON), notification_preferences (JSON).

Use Cases and Transition Graphs:

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.

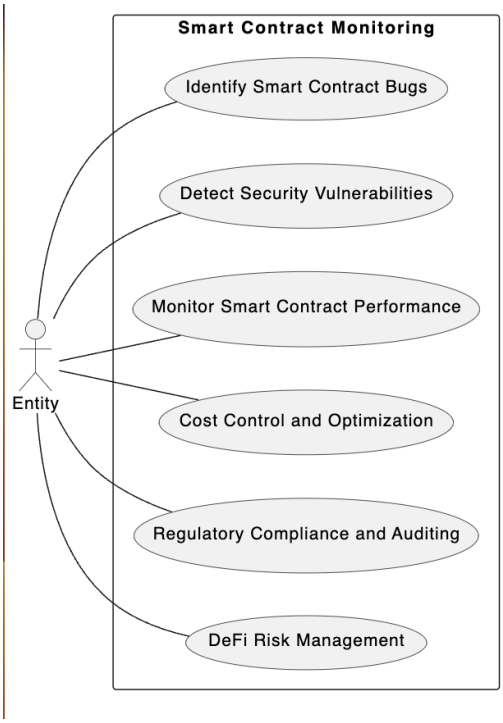
3. Monitoring Smart Contract Performance: If a contract suddenly starts using more gas for typical operations, it may be due to an inefficient code update or change in the contract's state, signaling the need for optimization.

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.

Use case Diagram:



The potential customers to get feedback for a gas usage anomaly detector are typically stakeholders in the blockchain ecosystem who interact with smart contracts and have a vested interest in their efficient and secure operation. This can include:

1. Smart Contract Developers and Auditors: They can use the detector during the development and auditing phases to optimize gas costs and ensure security best practices are followed.

2. Blockchain Project Teams: Teams that manage smart contract-based projects, such as tokens or decentralized applications (DApps), would use the detector to monitor their contracts post-deployment.

3. Decentralized Autonomous Organizations (DAOs): DAOs that govern decentralized protocols and platforms need to monitor contract efficiency and security, especially when proposing and voting on upgrades or changes.

4. Decentralized Finance (DeFi) Platforms: DeFi platforms that offer complex financial products and services built on smart contracts would need to monitor for anomalies to manage risk and prevent exploits.

5. Enterprise Clients: Companies that are integrating blockchain technology into their business processes would use the detector to oversee their blockchain transactions and ensure cost efficiency.

Tools and Technologies:

Category	Tools and Technologies
Web3 Libraries	web3.js, ethers.js
Data Storage	PostgreSQL, MongoDB
Backend Frameworks	Node.js, Flask, Django
Frontend Frameworks (for UI)	React, Vue.js, Angular
APIs for Alerting	Email services, communication platforms

References:

1. https://github.com/eth-brownie/brownie/issues/1653
2. https://docs.vyperlang.org/en/v0.2.14/testing-contracts-brownie.html
3. https://eth-brownie.readthedocs.io/en/stable/tests-pytest-intro.html
4. https://k0nze.dev/posts/python-estimate-ethereum-gas/
5. https://ethereum.stackexchange.com/questions/27452/how-to-estimate-gas-cost
6. https://www.sciencedirect.com/science/article/pii/S2096720923000234
7. https://www.youtube.com/watch?v=UISu1jFJOGc
8. https://www.youtube.com/watch?v=ceGCPVH6PNQ
9. https://docs.python.org/3/library/sqlite3.html
10. https://moralis.io/python-and-web3-a-web3-and-python-tutorial-for-blockchain-development/