Real Check

A BlockChain Based Fake Product Detection System

https://github.com/Rounaknayee/RealCheck

Rounak Nayee
(MS Computer Science)
Rutgers University - Camden

Abstract

In recent years, blockchain technology, exemplified by applications like Bitcoin, has garnered attention for its capacity to address issues like double-spending and ensure transaction legitimacy without centralization. This project explores a decentralized blockchain approach to empower consumers in verifying product authenticity and reducing reliance on merchants. By implementing a blockchain system for anti-counterfeiting, manufacturers can offer genuine products efficiently without managing direct stores, thereby reducing quality assurance costs. Counterfeit goods pose a global problem, eroding consumer trust. Leveraging blockchain's trust-building capabilities, this project tackles counterfeit product sales. Manufacturers can record authentic product serial numbers on the blockchain ledger, enabling consumers to verify product authenticity using these serial numbers before purchase. Blockchain secures data integrity and fosters a trusted environment. Counterfeit products jeopardize consumer safety, brand reputation, and revenue. Blockchain-based authenticity identification systems are essential to combat this issue, ensuring the detection and validation of genuine products, thus restoring consumer trust. QR codes play a crucial role. Scanning these codes connects to the blockchain, storing product details as database blocks. Matching codes confirm authenticity, ensuring consumers aren't solely dependent on retailers to verify product legitimacy.

Introduction

The issue of counterfeit products poses a persistent risk that can impact both companies and consumers on a global scale, particularly in industries such as medical supplies, electrical gadgets, and cosmetics. Detecting counterfeit products is not only challenging but also potentially dangerous for consumers. The consequences of counterfeit goods extend beyond immediate safety concerns, affecting a company's reputation, earnings, and the well-being of its customers. For example, reports from the Authentication Service Provider Association (ASPA) for 2021 indicated that the Indian economy could lose INR 1 trillion annually due to counterfeit products. Additionally, ResearchAndMarkets estimated that counterfeit goods amounted to a staggering 1.2 trillion USD in 2017, with projected harm reaching 1.82 trillion USD globally by 2020. This underscores the urgency of addressing the issue.

Counterfeit products, typically low-quality imitations of genuine items, have become increasingly difficult to distinguish from their authentic counterparts. According to the Organization for Economic Co-operation and Development (OECD), the global trade in counterfeit goods has steadily risen and now represents 3.3% of global trade. This illicit trade not only siphons revenue away from legitimate brands but can also endanger consumers' health, especially in industries such as medicine and beauty products.

Online retailers are taking measures to combat counterfeit products on their platforms. For instance, Amazon's Project Zero initiative, involving machine-learning technology and a substantial financial investment, has been successful in blocking the listing of counterfeit products and removing fake goods from its warehouses. However, the prevalence of counterfeit products has left consumers skeptical, with nearly 1 in 10 European Union consumers reporting they have unknowingly purchased counterfeit items, eroding trust in online purchases and affecting authentic brands.

In light of these challenges, there is a pressing need for a reliable method to authenticate products, especially in second-hand markets. Blockchain technology emerges as a solution, offering a platform of trust through distributed ledgers characterized by consensus, provenance, immutability, and finality. Blockchain can empower consumers to verify product authenticity, significantly reducing the sale of counterfeit goods and allowing consumers to shop with confidence.

This project is based on a comprehensive and irreplicable product anti-counterfeiting system based on Blockchain. The roles within this system include Manufacturers, Suppliers / Sellers, and Consumers. Manufacturers can add supplier addresses and monitor the information, while Suppliers can update the information about a product for consumers to confirm their authenticity. Consumers, in turn, can verify final sellers credibility and product availability. By implementing this Blockchain-based system, the project aims to provide an effective solution to the pervasive problem of counterfeit products, ultimately benefiting both consumers and manufacturers alike.

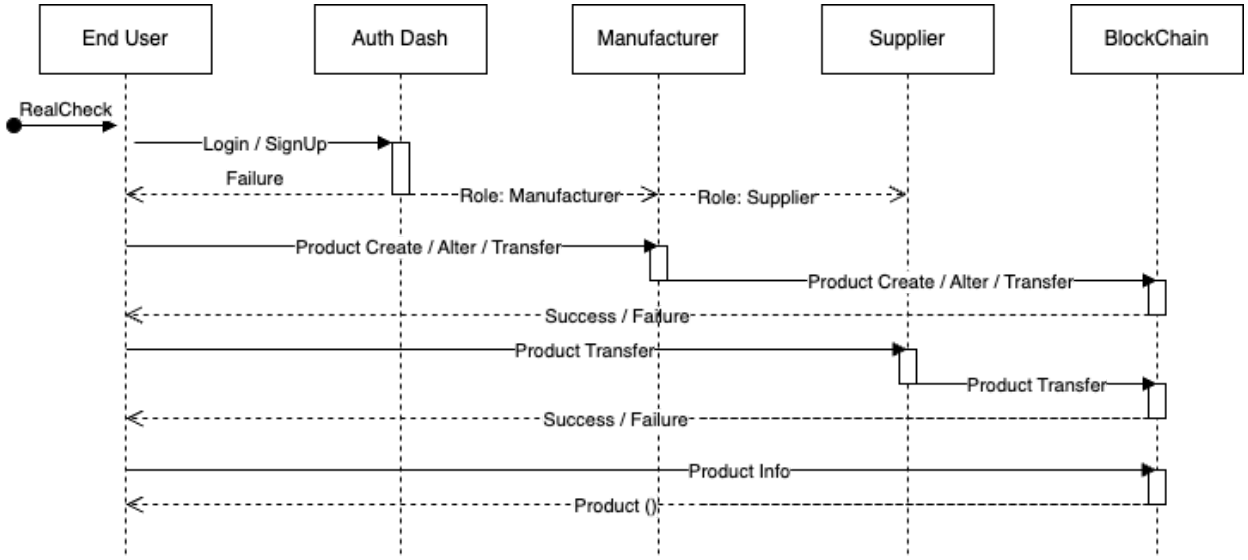
Table: The state-of-the-art comparison of the general supply chain and the proposed work.

S. No.	General Supply Chain	Proposed Blockchain based Supply Chain
1	Improper product handling with less focus on quality	Proper product handling with major focus on quality and authenticity
2	Tracking-and-tracing products is complex and challenging	Products can be easily tracked-and-traced in our blockchain environment
3	Less transparency and more cost expenditure in authenticating the products	More transparency and less cost expenditure in authenticating the products
4	No higher authorities involved in product and the supply chain members inspection	A “Quality Control Officer” is deployed to inspect the products and correct the supply chain members in case of discrepancies

Implementation

The Users interact with the system through a web-based interface. This frontend is designed for ease of use, providing a seamless experience for submitting requests and accessing information, developed with the help of React Framework. In operation, whenever a user submits a request, the frontend initiates the request to the backend. The user requests from the frontend are processed by a server built using Node.js and the Express framework. This server acts as the intermediary, handling requests and facilitating communication between the frontend and the blockchain/database as well as storing the user states and transactions.

The following Sequence diagram depicts the interaction of different users at various stages with the application:



The Realcheck's blockchain part is designed using Ethereum as the underlying blockchain platform, with smart contracts written in Solidity, Ethereum's high-level programming language. Solidity, tailored for the Ethereum Virtual Machine (EVM), supports features like inheritance and libraries importing and is Turing complete, offering more functionality than Bitcoin's scripting language.

In this implementation, the Sepolia testnet was selected over a private chain for a more realistic Ethereum environment. Sepolia provides an accurate simulation of the Ethereum mainnet, offering better testing conditions, community support, and ease of use. It allows for seamless integration with existing Ethereum tools and ensures stable, secure testing. This streamlines the development process, enhancing the reliability and robustness of smart contract deployment.

Alongside the blockchain, the system incorporates a MongoDB database for storing

non-blockchain data, including user sessions, and is managed by a Node.js server ensuring data integrity and security. For robust security, it employs JSON Web Tokens (JWT) for user authentication and authorization. Each user request from the frontend is authenticated via JWT, and user sessions are securely stored in MongoDB, providing a secure, stateless method for verifying user identities and permissions. This layered approach ensures that all interactions with the blockchain and database are legitimate and conform to high-security standards.

Let's further delve into the project's implementation which is primarily divided into 5 main components Viz, Frontend, Backend, Smartcontracts(Blockchain), Infrastructure, and the Database.

1. Frontend

The frontend is a crucial component designed as a single-page application (SPA) using React, a powerful JavaScript library that excels in building user interfaces with high interactivity and seamless user experiences. React's component-based architecture felt ideal for developing complex interactions in the application which required dynamic content updates without reloading the page, embodying the core principles of SPAs.

Also, the choice of React and JavaScript for the frontend development of the project is rooted in several strategic advantages that align with the project's goals:

Component-Based Architecture: React's modular nature allows for encapsulating components, making the code more manageable and reusable. This was ideal where components such as product listings, user authentication forms, and QR code generators were developed independently just once and then integrated at multiple places.

Interactive User Interfaces: JavaScript, with React, is adept at creating dynamic and responsive user interfaces. This aided in providing a smooth and engaging experience for users as they interacted with the system, which was crucial for operations like verifying product authenticity or managing transactions.

State Management: React's state management capabilities such as the redux store and UseEffects enabled the application to respond in real-time to user interactions and system changes, crucial for maintaining an up-to-date interface that reflects the current state of product data and user sessions.

Efficient Updates: React's virtual DOM optimizes rendering, ensuring that only components that have changed are updated. This leads to efficient performance, which is important for applications as it deals with real-time data such as product statuses and user information.

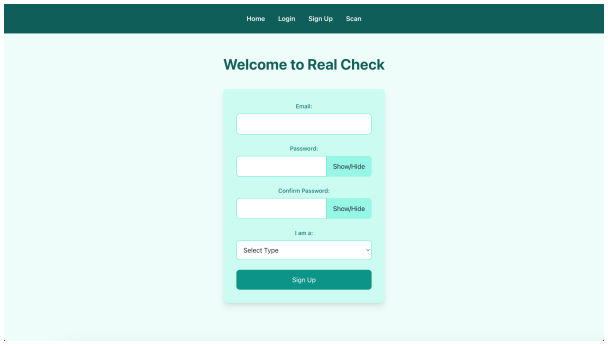
Strong Community and Ecosystem: JavaScript and React have extensive community support and a rich ecosystem of tools and libraries, which helped in providing a wealth of resources and pre-built solutions to common problems, streamlining the development process for application.

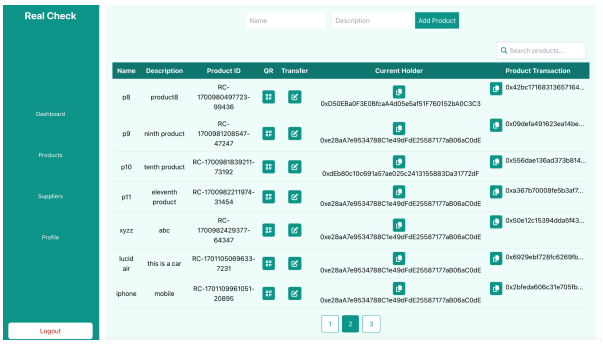
Apart from this, the chosen packages also played a significant role in the functionality and aesthetics of the frontend such as:

Axios: It enhances the React frontend by streamlining asynchronous HTTP requests with its promise-based structure, simplifying the handling of operations like retrieving product details or posting verification data. Its interceptors allow us to efficiently append authentication tokens and manage errors, ensuring consistent and secure communication throughout the app. Additionally, Axios's robust error-handling capabilities are crucial for gracefully managing any potential network or blockchain-related errors.

QR Code Generation (qrcode.react): This package allows the application to generate QR codes, a feature that links physical products to their digital blockchain records seamlessly.

React Router (react-router-dom): It manages the routing, enabling navigation between different views without page refreshes, essential for SPA behavior.





These figures depict the various important views such as the login and the products page which are the most frequently used parts of the application.

2. Backend

When deciding on the technology stack for the backend, I opted for a JavaScript-based solution using Node.js and the Express framework, over other languages like Python or Java, for several reasons:

1. Unified Language Ecosystem: Utilizing JavaScript for both frontend and backend streamlines development, allowing for a shared language and data types across the entire application stack. This unified ecosystem can lead to more efficient development and fewer context-switching errors.

2. Performance and Scalability: Node.js is renowned for its performance, especially in handling asynchronous operations and I/O-bound tasks, which are common in a service interacting with blockchain technologies. Its non-blocking I/O model ensures that the backend can handle a high number of simultaneous connections without a performance hit.

3. Community and Library Support: JavaScript has a vast community, and with Node.js being one of its most popular runtimes, there is extensive support available. The npm registry is filled with packages that were easily integrated into the project, such as ethers for interacting with the blockchain.

4. Blockchain Compatibility: The choice of JavaScript was particularly advantageous for blockchain-related tasks. Libraries like Ethers.js provide comprehensive functions to interact with Ethereum, smart contracts, and other blockchain protocols. This native compatibility and support helped streamline the process of executing blockchain transactions and querying blockchain data.

Along with these factors, the backend also incorporates several key packages:

Alchemy SDK: This simplifies the interaction with the Ethereum network, providing a set of tools to easily connect with nodes and send transactions.
Bcrypt.js: It secures user data by hashing and salting passwords, essential for maintaining data integrity and security.
CORS: This package is crucial for handling Cross-Origin Resource Sharing, allowing or restricting requested resources on a web server depending on where the HTTP request was initiated.
Dotenv: Manages environment variables, keeping sensitive credentials like API keys and database URIs secure.
Ethers.js: A complete Ethereum wallet implementation and utilities in JavaScript, allowing for interaction with the Ethereum Blockchain.
Express: A minimalist web framework for Node.js, facilitating the creation of fast and secure web applications and APIs.
JSON Web Tokens (JWT): Facilitates user authentication by generating tokens for session management.

Mongoose: An Object Data Modeling (ODM) library for MongoDB and Node.js that manages relationships between data, provides schema validation, and is used to translate between objects in code and their representation in MongoDB.

Backend works by employing special pathways—or 'routes'—that cater specifically to the different users interacting with our system: manufacturers, suppliers, and customers. Each of these routes is tailored to handle the unique actions they need to take, like creating new products or updating new products. To manage the data flow, I used simple HTTP methods; with POST methods typically handling data creation or updates, while GET methods are used for data retrieval.

A big part of making sure the backend runs smoothly and only authorized people interact with the system is using *'middleware'*. Think of it as a kind of checkpoint that every protected request has to pass through. I set up one type of middleware named *"Auth"* to check that anyone asking to perform an action is really authenticated into the system—this is done using JWTs, which are like secure passes. Whenever a user authenticates successfully, the backend sends back a JWT token while storing it within the database at the same time. Then at every subsequent request to the backend, this token is sent which is then used by the auth middleware to verify if the user is really authenticated and authorized to perform the specific actions before the action is performed in the backend. Another type of middleware function named *"CheckRole"* checks whether the user is allowed to do what they're asking to do, like whether they're actually a manufacturer or a supplier if the action requires being one.

In essence, the backend is engineered to be efficient, secure, and highly compatible with blockchain technologies, leveraging the strengths of JavaScript and the Node.js environment

3. Smartcontract (Blockchain):

The implementation of a smart contract on the blockchain serves as the cornerstone for establishing a robust level of trust among all parties involved in the supply chain. This trust is pivotal, especially in industries where the authenticity and traceability of products are critical. The smart contract automates various processes involved in the product lifecycle. For instance, it can automatically verify and log transactions without the need for intermediary verification, reducing the possibility of human error and increasing efficiency. By assigning a unique serial number to each product and recording its journey on an immutable ledger, it provides a transparent, reliable, and efficient system for tracking product authenticity and lineage. Following are the different components of the contract that help make up the system:

The ‘Product' structure is a key component of the contract and is defined as follows:

name (string): The name of the product.
manufacturer (string): The name of the manufacturer of the product.
currentHolder (address): The Ethereum address of the current holder or owner of the product.
chainOfCustody (address[]): An array of Ethereum addresses representing the history of holders of the product, from the manufacturer to the current holder.
isFinalized (bool): A boolean indicating whether the product's information and ownership chain is finalized and cannot be altered further.
manufacturerAddress (address): This is a 20-byte value that represents an Ethereum address. The Ethereum address of the manufacturer who initially created the product entry.

Mapping: A key-value pair data structure. mapping(string => Product) maps a string (product ID) to a Product struct, allowing efficient lookup of product details using their ID.

```
struct Product {
    string name;
    string manufacturer;
    address currentHolder;
    address[] chainOfCustody;
    bool isFinalized;
    address manufacturerAddress;
}
```

The various Functions within contract:

1. addProduct

Purpose: To add a new product to the blockchain.
Inputs: productId (unique identifier for the product), name (product's name), manufacturer (manufacturer's name).
Functionality: Initializes a new Product struct and sets its initial values, including marking the sender (msg.sender) as the initial holder and manufacturer. It also flags the product as not finalized (isFinalized set to false).

```
function addProduct(string memory productId, string memory name, string memory manufacturer) public {
    Product storage product = products[productId];
    product.name = name;
    product.manufacturer = manufacturer;
    product.currentHolder = msg.sender;
```

```
    product.chainOfCustody.push(msg.sender);
    product.manufacturerAddress = msg.sender;
    product.isFinalized = false;
    emit ProductAdded(productId, msg.sender);
}
```

2. alterProductHolder

Purpose: To change the holder of a product.
Inputs: productId (product's unique identifier), newHolder (address of the new holder).
Preconditions: Only the manufacturer of the product can change the holder.
Functionality: Updates the currentHolder of the product and appends the new holder to the chainOfCustody.

```
function alterProductHolder(string memory productId, address newHolder) public{
    Product storage product = products[productId];
    require(msg.sender == product.manufacturerAddress, "Only the manufacturer can update this product");
    product.currentHolder = newHolder;
    product.chainOfCustody.push(product.currentHolder);
    emit ProductHolderChanged(productId, newHolder);
}
```

3. transferProduct

Purpose: To transfer a product from its current holder to a new one.
Inputs: productId, nextHolder (address of the next holder).
Preconditions: The sender must be the current holder, and the product must not be finalized.
Functionality: Similar to alterProductHolder, but used for transferring between non-manufacturer parties. Updates currentHolder and modifies chainOfCustody.

```
function alterProductHolder(string memory productId, address newHolder) public{
    Product storage product = products[productId];
    require(msg.sender == product.manufacturerAddress, "Only the manufacturer can update this product");
    product.currentHolder = newHolder;
    product.chainOfCustody.push(product.currentHolder);
    emit ProductHolderChanged(productId, newHolder);
}
```

4. finalizeProduct

Purpose: To finalize the product information, preventing further changes.
Inputs: productId.
Preconditions: Only the current holder can finalize the product.
Functionality: Sets isFinalized to true, indicating the product information and ownership chain cannot be altered.

```
function finalizeProduct(string memory productId) public {
    Product storage product = products[productId];
    require(msg.sender == product.currentHolder, "Only the current holder can finalize this product");
    product.isFinalized = true;
    emit ProductFinalized(productId, msg.sender);
}
```

5. getProductChain

Purpose: To retrieve the chain of custody of a product.
Inputs: productId.
Output: Returns the chainOfCustody array for the specified product.

```
function getProductChain(string memory productId) public view returns (address[] memory) {
    return products[productId].chainOfCustody;
}
```

6. getProduct

Purpose: To get detailed information about a product.
Inputs: productId.
Output: Returns all details of the specified product, including name, manufacturer, current holder, chain of custody, finalized status, and manufacturer's address.

```
function getProduct(string memory productId) public view returns (string memory, string memory, address, address[] memory, bool, address) {
    Product storage product = products[productId];
        return (product.name, product.manufacturer, product.currentHolder, product.chainOfCustody, product.isFinalized, product.manufacturerAddress);
}
```

In all of the above function implementations lets understand **‘msg.sender’**

msg.sender is a global variable in Solidity that represents the address of the entity (person or contract) that is currently calling the function. It's crucial for determining the context of transactions and enforcing permissions. In the ProductAuthenticity smart contract, msg.sender is used in several contexts:

Identifying the Manufacturer: When a product is added using addProduct, msg.sender is stored as manufacturerAddress, signifying who created the product entry.

Authentication and Authorization: In functions like alterProductHolder, transferProduct, and finalizeProduct, msg.sender is used to check whether the entity calling the function has the right to perform that action. For example, in transferProduct, it checks if msg.sender is the current holder of the product to make changes within the particular product..

Tracking Product Custody: When a product is transferred, msg.sender is used to update the chainOfCustody, ensuring that the transfer history of the product is accurately recorded.

In essence, msg.sender is a fundamental aspect of Solidity smart contracts that helps in identifying the initiator of a transaction and is critical for implementing access control and maintaining the integrity of transactions.

Also, certain datatypes are being used in the contract as follows:

memory: This is a temporary place to store data. It’s erased between external function calls and is cheaper to use.
storage: This is where the contract state variables are held. It’s persistent between function calls and transactions but is more expensive in terms of gas costs.
public view: This function is one that can be called externally and internally, and it promises not to modify the state of the contract. It's used for getter functions which aid in returning the state variables of the contract.

The above contract for our project was deployed on Sepolia Tesnet using Remix IDE at Contract address “**0xE5D902c3b852aeaf3D34Df8036bf3013dB275e28**” and can be accessed at this Link.

For interacting with the blockchain, the smart contract helper class is used by the backend which takes a private key, crucial for signing transactions and authenticating the user on the blockchain. By loading the contract's Application Binary Interface (ABI) from a JSON file, the function prepares to accurately communicate with the smart contract, adhering to its defined interface. Connection to the Ethereum network is established through Alchemy provider; instantiated via JsonRpcProvider, acting as a node and facilitating interactions with the Ethereum network, allowing for the sending of transactions and querying of blockchain data. The combination of the

provider with the user's private key creates a Wallet instance. This instance not only manages the user's Ethereum account but also acts as a signer, enabling the execution of transactions with the necessary security and authority.

4. Database

The application has been designed using a MongoDB (NoSQL) database, for storing specific data models in the application. It excels in managing semi-structured data, the type used by applications. MongoDB's flexibility allows for easy modifications and additions to the database schema without affecting existing data, accommodating the evolving nature of the application. The data models within the application include varied data types like strings, arrays, and nested objects, that can be more effectively handled in a NoSQL environment. It also provides horizontal scalability, which can be essential for handling large volumes of data and high-traffic loads efficiently. The decision to use MongoDB in conjunction with the blockchain is not only a technical choice but also a cost-efficient strategy. Storing all data on the blockchain can be prohibitively expensive due to transaction fees, especially for non-critical or large data sets. Storing only crucial parts of the data, such as transactional records or blockchain-specific information, on the blockchain, and keeping the remaining metadata in MongoDB, aids in achieving a balance between the immutable record-keeping of blockchain and the efficient, cost-effective data management of a traditional database. The two primary types of models used in the database are Products and Users.

Users Model:

The users model comprises Email, Password, Role, Wallet Address, Private Key, Public Key, and Session Tokens where the email is a unique field for each user to avoid duplicity and the role defines the manufacturer or supplier to reference to the correct authorizations for use in the application. This model is designed to manage user authentication, authorization, and wallet-related information, essential for the blockchain interactions within application.

Products Model:

This model includes fields such as Name, Description, Product ID, Product Transaction, Manufacturer Address, Manufacturer Email, and Current Holder. These fields provide comprehensive information about each product, which is crucial for tracking and management within the supply chain as well as within the application.

5. Infrastructure:

It embraces a blend of the latest in cloud technology and DevOps best practices to create a

consistent highly scalable and performant application. At the heart of the system lies the powerful combination of Docker and Amazon Web Services (AWS) through ECS container orchestrator, providing a reliable and consistent environment for deployment. The strategic use of GitHub Actions for continuous integration and continuous deployment (CI/CD), along with Terraform for infrastructure as code, significantly streamlines the deployment process into the workflow. This approach not only enhances the efficiency and reliability of the system but also ensures a high degree of reproducibility in the deployment phases. The detailed components within the infrastructure can be explained as follows:

Frontend Deployment on AWS S3

Upon compilation, the React application is hosted on Amazon S3, a scalable and reliable object storage service. This hosting choice provides high availability and durability, ensuring that the frontend is always accessible to users.

Backend Deployment on AWS ECS

The backend application is containerized using Docker, which encapsulates the application and its dependencies in a Docker container. This containerization ensures consistent environments across development and deployment. The dockerized backend is then pushed by the CI/CD Pipelines to be hosted on Amazon Elastic Container Service (ECS), which is a highly scalable and fast container management service, offering features like automated scaling and load balancing.

GitHub Actions Workflow

The Continuous Integration and Continuous Deployment (CI/CD) pipeline is implemented using GitHub Actions. This workflow automates the processes of building, testing, and deploying both the frontend and backend applications. It is configured to trigger automatically on pushes to the main branch, specifically monitoring changes in the backend/ and frontend/ directories. Additionally, the workflow supports manual triggers, providing flexibility in deployment management. This setup ensures that the latest changes are always deployed efficiently, improving the overall deployment cycle.

Terraform Infrastructure as Code

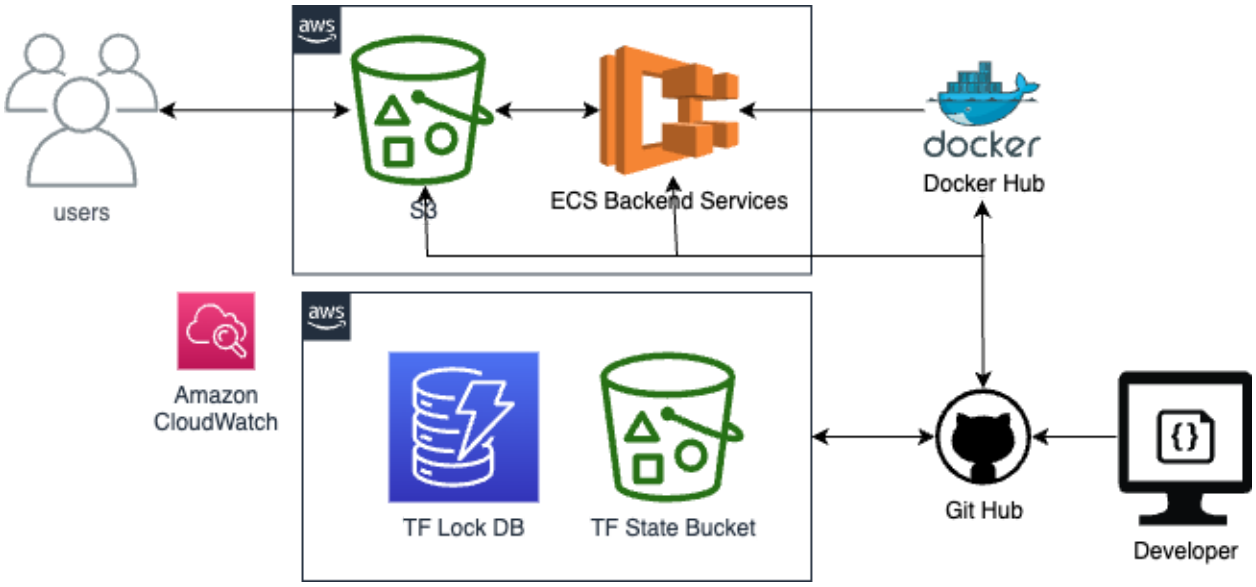
Infrastructure management is handled using Terraform, an Infrastructure as Code (IaC) tool. Terraform allows for the declarative configuration of cloud resources, making infrastructure deployments reproducible and predictable. The Terraform code specifies the necessary AWS resources, such as the ECS cluster (aws_ecs_cluster), task definition (aws_ecs_task_definition),

CloudWatch log group (aws_cloudwatch_log_group), ECS service (aws_ecs_service), and the network load balancer (aws_lb). It also defines the target groups, listeners, and security groups, ensuring a comprehensive setup of the ECS environment.

Security and Network Configuration

AWS security groups are configured to control inbound and outbound traffic to ECS service, enhancing the security of the application. An AWS network load balancer is used to distribute incoming backend application traffic across multiple targets, increasing the availability of the application.

The following diagram depicts RealCheck's implemented System Architecture:



Challenges faced:

1. Testing Smart Contracts for MVP

Testing smart contracts for the MVP on a blockchain, especially on testnets like Sepolia, is a critical and challenging step. It involves setting up simulated environments and optimizing smart contracts for performance and security. This testing phase was crucial to ensure the smart contracts functioned correctly and were free from bugs/vulnerabilities before deployment on the live network.

2. Learning and Implementing Solidity Contracts

Adapting to Solidity for smart contract development involved a steep learning curve. The challenge included mastering Solidity syntax, deploying contracts on testnets like Sepolia to

minimize costs, and balancing functionality with transaction costs. Understanding these aspects is essential for developing secure and efficient smart contracts on the Ethereum network.

3. Implementing React Routing
Implementing React Routing in single-page applications (SPAs) is challenging. It requires understanding how to manage navigation and component rendering based on URL changes, as well as handling state management across different views. Effective use of React Router was the key to ensuring a consistent and responsive user interface in SPAs.

4. Secure Session Management
Managing secure user sessions in the backend involved using authentication tokens, like JWTs, and protecting them from web vulnerabilities. The challenge lies in developing a robust session management system that efficiently handles the creation, validation, and termination of sessions, ensuring the security of user data throughout the application.

5. Infrastructure Choice and Cost Management
Choosing the right infrastructure for the project required evaluating various technologies and their associated costs. This decision-making process involved analyzing the trade-offs between different databases, backend languages, and cloud services. Selecting the right combination of technologies was crucial for achieving a balance between performance, scalability, cost-effectiveness, and maintainability of the application.

Future Scope:

1. Enhancing Product Finalization for End Consumers
Refining the process of recording product details on the blockchain at the finalization stage would allow end consumers to make a future resale of the product, especially in the case of precious products such as diamonds and gold which can last indefinitely. This would ensure every critical piece of product information is captured and easily verifiable on the blockchain itself including the metadata, enhancing transparency and authenticity as well as longevity of the blockchain's data integrity.

2. Building Flexible Smart Contracts
Develop smart contracts that can be updated as needed, allowing for seamless integration of new features and adaptations to changing requirements. This approach will ensure that our blockchain solutions remain both current and scalable.

3. Encouraging Consumer Participation
I envision a system where consumers play an active role in verifying product authenticity. By offering incentives for reporting counterfeit goods, I hope to foster a community-driven approach

to maintaining product integrity. Additionally, implementing mechanisms for manufacturers to receive and act on such reports efficiently creates a responsive and trustworthy ecosystem.

Conclusion:

In summary, this project has developed a blockchain-based system aimed at curtailing the sale of counterfeit products. With manufacturers and sellers as the key transaction participants, the system enables the recording of authentic products and their subsequent transition within the supply chain all the way upto the final sale to the end consumer. This ensures the traceability of each item, providing assurances to consumers about the authenticity of their purchases, which is especially valuable for items with enduring value and resale potential.

The project, thus effectively lowers the barrier to anti-counterfeiting measures, offering a more accessible means for companies to protect their products and for consumers to avoid counterfeit goods. This endeavor not only addresses brand protection but also enhances consumer confidence in the marketplace.

References / Resources

1. Shubham Prajapati, Jayesh Gadhari, Tushar Sawant, Juilee Kini, Sheetal Solanki, "Strengthening Supply Chain Integrity with Blockchain-based Anti-Counterfeiting Measures", 2023 International Conference on Innovative Data Communication Technologies and Application (ICIDCA), pp.786-790, 2023. https://ieeexplore.ieee.org/document/9766493
2. S. Prajapati, J. Gadhari, T. Sawant, J. Kini and S. Solanki, "Strengthening Supply Chain Integrity with Blockchain-based Anti-Counterfeiting Measures," 2023 International Conference on Innovative Data Communication Technologies and Application (ICIDCA), Uttarakhand, India, 2023. https://ieeexplore.ieee.org/document/10100264
3. A Blockchain-Based Application System for Product Anti-Counterfeiting https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8985337
4. Shivam Singh, Gaurav Choudhary, Shishir Kumar, Vikas Sihag and Arjun Choudhary, Counterfeited Product Identification in a Supply Chain using Blockchain Technology, August 2021.

https://a6c9b63e-5945-41f5-a670-61b4c439dc72.usrfiles.com/ugd/a6c9b6_e386e4bbd2204639b9f749d8c4e3f799.pdf

5. FAKE PRODUCT DETECTION USING BLOCKCHAIN TECHNOLOGY
   Kishan Tiwari*1, Nikita Patil*2, Akshay Gupta*3, Akash Sabale*4, Vina Lomte*5
   https://www.irjmets.com/uploadedfiles/paper/issue_5_may_2023/41058/final/fin_irjmets1685631927.pdf
6. J. Ma, S. -Y. Lin, X. Chen, H. -M. Sun, Y. -C. Chen and H. Wang, "A Blockchain-Based Application System for Product Anti-Counterfeiting," in IEEE Access, vol. 8, pp. 77642-77652, 2020.
   https://ieeexplore.ieee.org/document/8985337
7. https://react.dev/
8. https://reactrouter.com/en/main
9. https://docs.soliditylang.org/en/v0.8.23/
10. https://expressjs.com/
11. https://chat.openai.com/
12. https://docs.github.com/en/actions
13. https://www.terraform.io/
14. https://registry.terraform.io/providers/hashicorp/aws/latest/docs
15. https://hub.docker.com/
16. https://sepolia.etherscan.io/
17. https://docs.soliditylang.org/en/v0.8.23/