

VOCA Documentation - Technical Guide

Project Name: VOCA

Authors: Zak Smith, Senan Warnock

Supervisor: Geoffrey Hamilton

Table of Contents

1. Introduction
2. Design
3. Implementation
4. Sample Code
5. Problems Solved
6. Results
7. Future Work

Abstract

The VOCA project presents a zk rollup solution aimed at enhancing the scalability and efficiency of Ethereum transactions. This innovative approach utilises zk circuits and smart contracts to execute Ethereum and ERC-20 transactions off-chain, significantly reducing the load on the main Ethereum network while ensuring transaction validity and security through zero-knowledge proofs. Although the implementation of the operator node remains incomplete, the zk circuits and proofs have been successfully validated, demonstrating the potential of the technology to improve transaction throughput without compromising security. The operator node, once completed, is designed to process deposit events and generate proofs of exclusion, updating the state of the rollup and handling user transactions by creating and posting zk proofs to the Ethereum mainnet for verification. This document serves as a comprehensive guide for further development and usage of the VOCA system, detailing both achieved results and areas requiring future work.

1. Introduction

Overview

The VOCA documentation serves as an all-encompassing manual for those interested in the architectural, operational, and functional facets of the VOCA project.

Purpose of the Documentation

- **Educational Insights:** To elucidate the cryptographic underpinnings and strategic objectives driving VOCA.
- **Technical Specifications:** To provide a granular view of VOCA's system architecture, covering both front-end and back-end frameworks.
- **User Guidance:** To offer comprehensive tutorials and guidelines that aid users and developers in navigating and utilising the system effectively.

By the conclusion of this guide, readers should gain an understanding of VOCA's operational mechanics, user interaction protocols, and avenues for community contribution.

2. Design

High-Level Overview

VOCA utilises a modern stack designed to enhance the scalability and efficiency of transactions within the Ethereum blockchain ecosystem. This section provides a detailed overview of both the frontend and backend components, their integration with blockchain and zk-rollup technologies, and the interaction flow ensuring a seamless user experience.

Frontend Architecture

- Framework & Languages: VOCA uses Next.js 14, TypeScript, JavaScript to develop a robust and scalable application.
- Blockchain Interaction: The integration with Ethereum is managed through Wagmi and ethers.js, which simplify blockchain interactions and provide easier access to blockchain functionalities.
- Interface: The responsive user interface is designed for optimal navigation and accessibility across different devices, enabling users to manage transactions and interact with the blockchain efficiently.

Backend Architecture

- Server Environment: The backend is powered by Node.js, which is optimised for fast, non-blocking data operations, suitable for high-performance backend services.
- Smart Contracts: Written in Solidity, these contracts handle all blockchain-based transactions and state changes, playing a crucial role in the system's operation.
- Circuits: The circuits are written in Circom, a powerful language for writing zk circuits.
- SnarkJS is used to generate the ZK Proofs.

Interaction Flow

The interaction flows of VOCA highlight the step-by-step processes involved in typical use cases like wallet connection and funds deposit, providing clear insights into user actions from start to finish. Each flow is supported by sequence diagrams that visually represent the process, helping both developers and users navigate and utilise the platform effectively.

Use Case 1: Connecting a Wallet in a Web3 Application

Goal:

The user connects their blockchain wallet to the web application to securely interact with blockchain features.

Actors:

- End User: A visitor to the platform intending to use blockchain functionalities.
- Web3 Modal: A library that provides a user interface for choosing and connecting to different wallets.
- Ethers.js: Libraries that facilitate interaction with the Ethereum blockchain.

Preconditions:

- The user has internet access.
- The user has a compatible wallet installed (e.g., MetaMask, WalletConnect-supported wallets).

Main Flow:

1. **User Initiates Connection:** The user clicks the "Connect Wallet" button on the web application, typically located in the navbar or on the landing page. This action triggers a function in the `ConnectButton.tsx` component.
2. **Display Wallet Options:** The application, via `web3modal.tsx`, presents a modal or overlay that lists available wallet providers. This modal supports various wallet types, including browser extensions (like MetaMask) and mobile wallet apps.
3. **User Selects Wallet Provider:** The user selects their wallet provider from the modal, which triggers a connection request to the wallet, managed by `web3modal.tsx`.
4. **Wallet Authorization:** The wallet prompts the user to authorize the connection to the web application.
5. **Establish Connection:** Upon user authorization, `web3modal.tsx` establishes a connection to the wallet. The wallet provider returns a provider object to the application, which is handled by `Ethers.js` to create a signer or provider instance.
6. **Update Application State:** The `ConnectButton.tsx` component updates the application's state to reflect the successful wallet connection. The UI updates to show the user's wallet address and possibly their balance as fetched from the blockchain.
7. **Enable Blockchain Interactions:** With the wallet connected, the user can now initiate blockchain transactions, such as those with our rollup smart contracts. These transactions can be initiated from other parts of the application like a "Send Transaction" and "Deposit" forms.

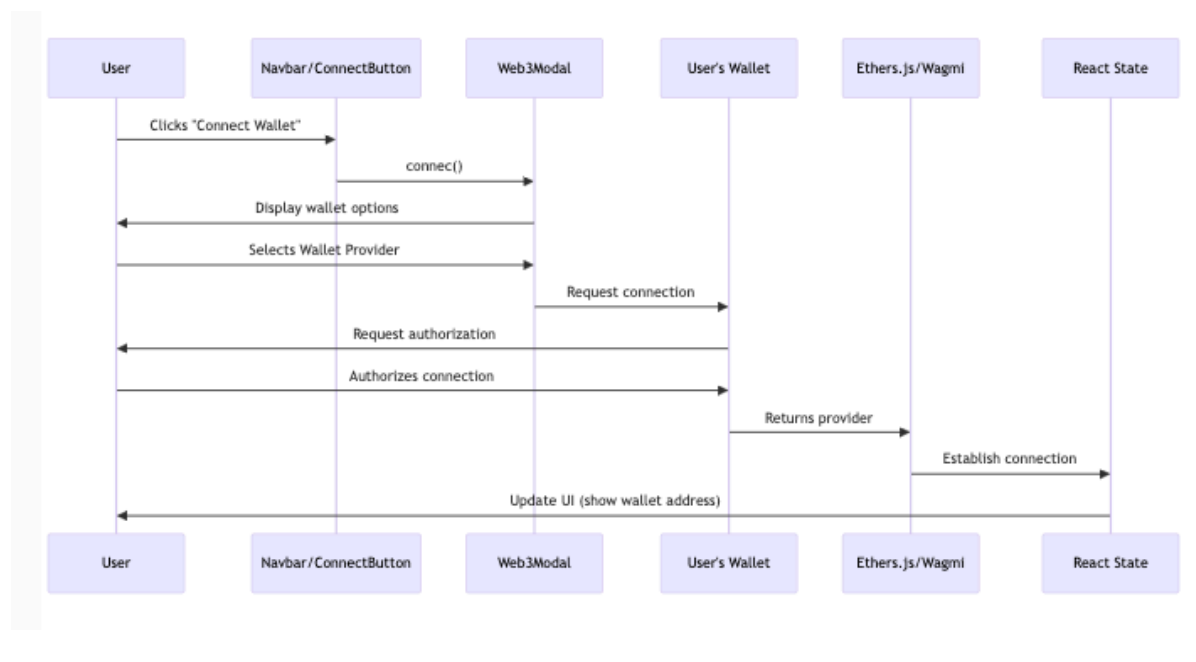
Alternative Flows:

- **Wallet Connection Denied:** If the user denies the connection request in their wallet, the application remains in its initial state, and the user is informed that they need to connect a wallet to use blockchain features.
- **No Wallet Found:** If `web3modal` detects no compatible wallet, it informs the user that they need to install a wallet to proceed.

Postconditions:

The user's wallet is connected, allowing them to interact securely with blockchain features provided by the application.

Sequence Diagram:



Use Case 2: Depositing Funds on the Frontend

Goal:

Enable users to securely deposit funds into their accounts on the rollup via the VOCA frontend.

Actors:

- End User: A user of the VOCA platform who wants to deposit funds.
- Wallet Service: Handles wallet operations including initializing and managing blockchain interactions.
- Deposit Component: Manages the UI and logic for deposit transactions.

Preconditions:

- The user is registered and has a digital wallet compatible with the platform (e.g., MetaMask).
- The user has already connected their wallet to the VOCA platform.

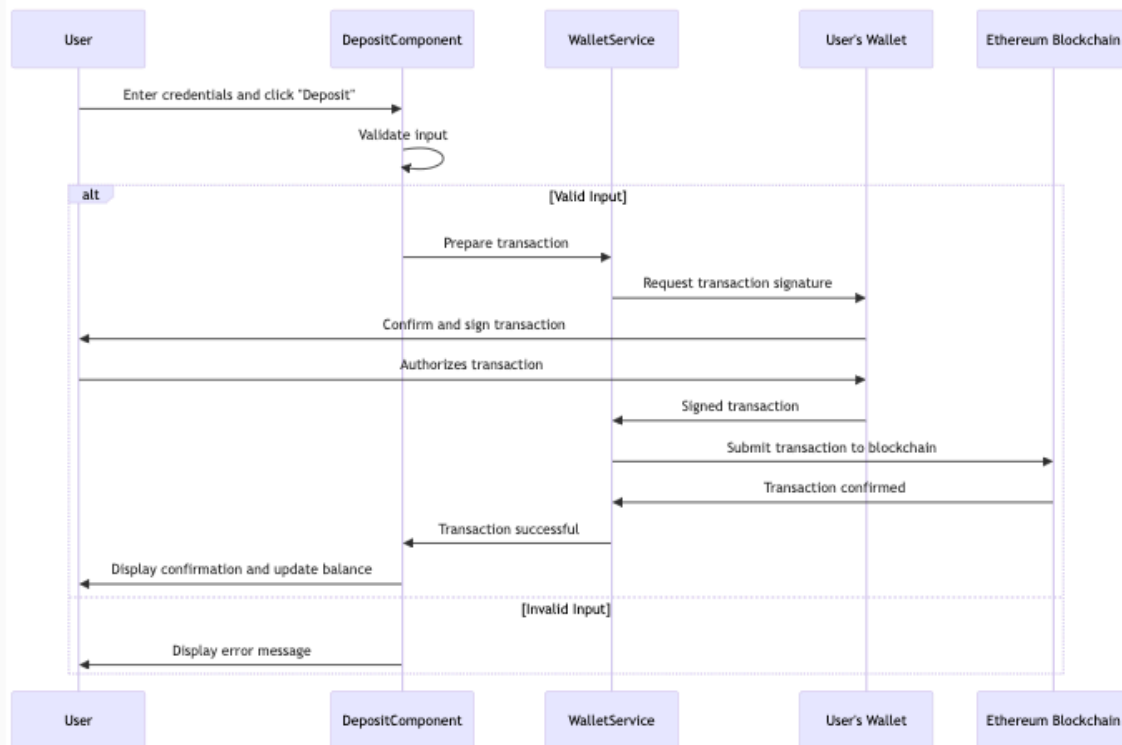
Main Flow:

1. Access Deposit Functionality: The user navigates to the deposit section of the platform, accessed via the dashboard.
2. Enter Deposit Details: In the UI, the user enters the amount they wish to deposit. This will be used in the DepositComponent. The component includes form fields for specifying the amount and possibly selecting the type of currency or token to deposit.
3. Initiate Deposit Request: The user submits the deposit request by clicking the "Deposit" button. This triggers the handleDeposit function, which prepares the transaction data.
4. Generate Transaction: The WalletService is invoked, which utilizes the Ethereum library (ethers.js) to create a transaction. This involves specifying the amount and the recipient address (the platform's deposit address).
5. Submit Transaction to Blockchain: The transaction is signed using the user's private key, which is managed by their browser wallet. The signed transaction is submitted to the network for processing.
6. Confirmation and Receipt: The user receives feedback on the status of the transaction through the UI. If successful, the transaction details are displayed. The user's balance is updated accordingly.

Postconditions:

The user's funds are securely deposited into their account on the blockchain. The user can view the updated balance and transaction history on their dashboard.

Sequence Diagram



Use Case 3: Sending a Transaction on the Frontend

Goal:

Enable users to securely send transactions through the frontend interface on the zk rollup.

Actors:

- End User: A user who intends to send transactions via the platform.

Preconditions:

- The user must have internet connectivity.
- The user's digital wallet must be connected, and they must have a valid public and private key pair available.
- The user must have sufficient funds to cover the transaction and any associated fees.

Main Flow:

1. **Access Transaction Feature:** The user navigates to the transaction section of the platform accessed from the dashboard.
2. **Input Transaction Details:** The user enters the recipient's public key and the amount they wish to send into the input fields provided in the TransactionComponent.
3. **Initiate Transaction:** Upon entering all required details, the user clicks the "Send" button. The system validates the input data, ensuring that all necessary fields are filled and the amount does not exceed the user's available balance.
4. **Generate Transaction Signature:** The system uses the user's private key and the transaction details to generate a cryptographic signature, ensuring the authenticity and non-repudiation of the transaction.
5. **Submit Transaction to Backend:** The fully formed transaction, including the signature, is sent to the backend server for further processing. The backend interacts with the blockchain to submit the transaction.
6. **Feedback and Confirmation:** Upon successful submission, the frontend receives a confirmation from the backend and notifies the user that the transaction has been successfully sent. If the transaction fails (e.g., due to an HTTP error), the user is informed of the failure and provided with the error message.

Alternative Flows:

- **Insufficient Funds:** If the user does not have enough balance to cover the transaction and fees, the frontend prevents the transaction from being sent and informs the user accordingly.
- **Invalid Input:** If the input validation fails (e.g., invalid recipient address), the user is prompted to correct the input data.
- **Backend Error Handling:** If the backend encounters issues such as blockchain congestion or operational errors, it returns an error response, which is handled by the frontend and displayed to the user.

Postconditions:

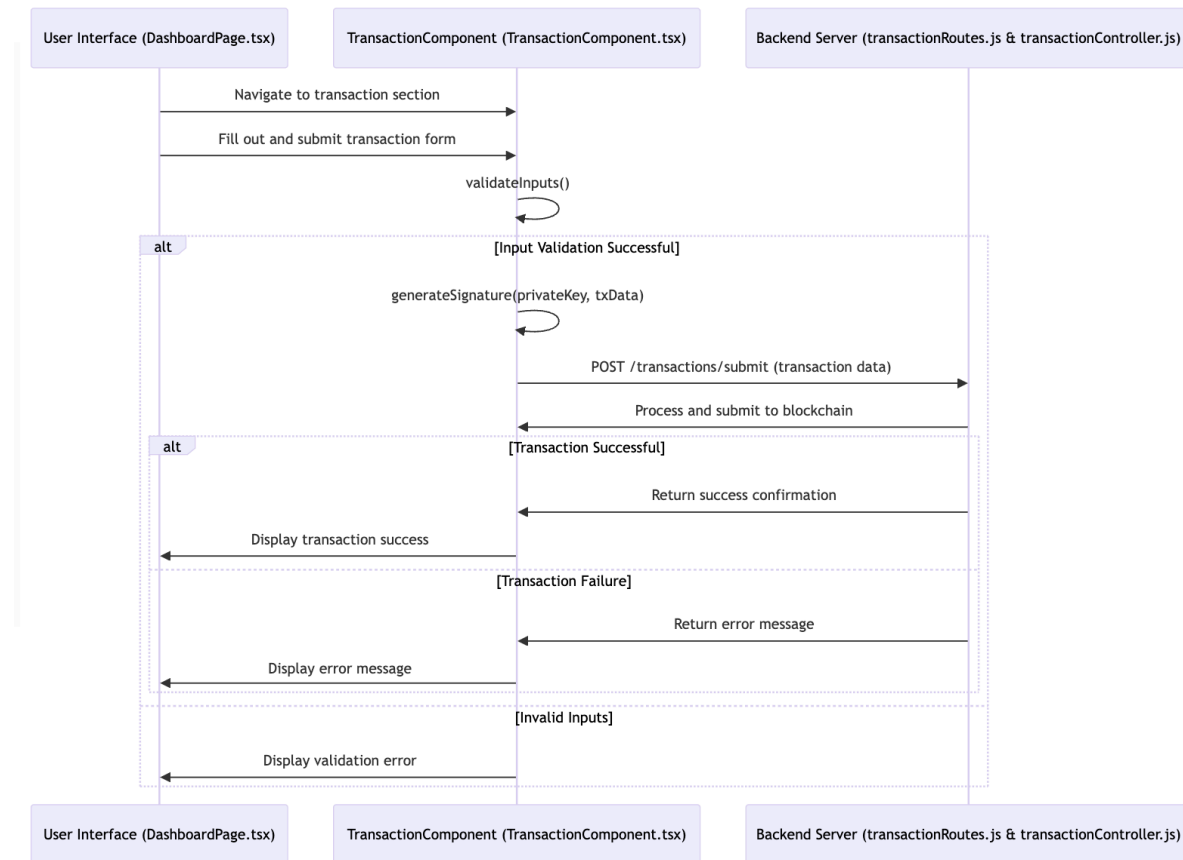
- **On successful transaction:** The user's balance is updated. The transaction is recorded on the blockchain. The user can view the transaction details, including the transaction hash, in their transaction history.
- **On failure:** No changes are made to the user's balance. The user is informed of the reason for failure.

Exception Handling:

- **Security Checks:** The system validates all transactions to ensure they are secure and the data integrity is maintained.

- **Error Handling:** The frontend provides clear and actionable error messages for issues such as network failures, invalid data, or backend service unavailability.

Sequence Diagram:



3.Implementation & Sample Code

This section of the documentation looks into the technical architecture of the VOCA project, covering both frontend and backend implementations. By detailing the code snippets, and operational logic.

This section is useful for those looking to contribute to the project, extend its features, or simply gain a better understanding of its technical foundation, this section serves as your essential guide.

Connecting a Wallet in a Web3 Application

The wallet connection functionality in the VOCA project is implemented using a `Web3Provider` component, which is crucial for integrating the blockchain capabilities seamlessly into the frontend. This component utilises `Web3Modal` for handling user interactions related to wallet management and `ethers.js` for blockchain interactions.

Web3Provider Component Setup

File: web3modal.tsx

The `Web3Provider` component manages the blockchain interactions within the VOCA platform. It performs several critical functions:

Initial Setup:

- Initializes `Web3Modal` with specific provider options to support various wallets (e.g., MetaMask, WalletConnect).
- Configures the network settings (e.g., Ethereum mainnet) and caching strategies to enhance user experience by allowing the wallet connection to persist across sessions.

Connect Function:

- Handles the user's request to connect their wallet. It triggers the UI for wallet selection and manages the authentication and connection process.
- Upon successful connection, it establishes a `Web3Provider` from `ethers.js`, facilitating further blockchain interactions like transactions or querying account details.

```
// Handles connecting the wallet through Web3Modal.
const connect = async () => {
  if (!web3Modal) {
    console.error("Web3Modal instance not initialized");
    return;
  }
  try {
    const instance = await web3Modal.connect(); // Connects to the selected wallet.
    const provider = new ethers.providers.Web3Provider(instance);
    const signer = provider.getSigner(); // Gets the signer from the provider.
    const account = await signer.getAddress(); // Fetches the connected account address.

    setProvider(provider);
    setAccount(account);
  } catch (e) {
    console.error("Could not connect to wallet:", e);
  }
};
```

Disconnect Function:

- Allows the user to disconnect their wallet. This function clears the cached session to ensure that subsequent connections require explicit user authentication.
- It's crucial for maintaining security, particularly in scenarios where the device is shared or public.

```
// Handles wallet disconnection.
const disconnect = async () => {
  if (!web3Modal) {
    console.error("Web3Modal instance not initialized");
    return;
  }

  try {
    await web3Modal.clearCachedProvider(); // Clears any session so wallet needs to reconnect.

    // If the provider can be disconnected, do it.
    const anyProvider = provider as any; // Using `any` to cover all provider types.
    if (
      anyProvider?.disconnect &&
      typeof anyProvider.disconnect === "function"
    ) {
      await anyProvider.disconnect();
    }
    setProvider(null);
    setAccount(null);
  } catch (e) {
    console.error("Failed to disconnect:", e);
  }
};
```

Depositing Funds on the Frontend

File: `DepositComponent.tsx`

The `DepositComponent` is a React functional component that provides users with an interface to deposit handled in the `handleDeposit()` function.

Key Features of the `DepositComponent`:

- **Interactive User Interface:** Provides input fields for amount and token type, and a button to submit the deposit.
- **Dynamic Error Handling:** Displays errors directly to the user if any issues occur during the deposit process.
- **Contract Interaction:** Uses a smart contract for depositing funds, which requires the contract to be loaded and connected before any actions can be taken.

Breakdown of `handleDeposit()`:

- **Validation check:** Ensures that all necessary fields are filled before attempting to execute the transaction. This includes checking if the deposit amount, public key, token type, and the contract are defined.

```
// Check all fields are filled
if (!depositAmount || !publicKey || !tokenType || !contract) {
  setError("All fields must be filled and a contract must be loaded.");
  return;
}
```

- **Public Key Handling:** Validates the public key components to ensure they are within uint256. This is critical to prevent errors during the deposit.

```
// Convert string publicKey parts to BigNumber to validate them
try {
  const x = ethers.BigNumber.from(publicKey.x);
  const y = ethers.BigNumber.from(publicKey.y);

  if (
    x.gt(ethers.constants.MaxUint256) ||
    y.gt(ethers.constants.MaxUint256)
  ) {
    setError("Public key parts are out of bounds for uint256");
    return;
  }
}
```

- **RollupContract Call:** the deposit is then sent with the parameters specified in the rollup contract to process the deposit on the rollup.

```
const transactionResponse = await contract.deposit(
  [publicKey.x, publicKey.y], // Pass the publicKey as an array directly
  ethers.utils.parseUnits(depositAmount, "ether"), // Convert deposit amount to wei
  tokenType,
  { value: ethers.utils.parseUnits(depositAmount, "ether") }
);
```

4. Problems Solved

The VOCA project has been designed to address a variety of challenges commonly faced in the Ethereum blockchain ecosystem, particularly those related to scalability, efficiency, and usability. A key aspect of our solution is the incorporation of rigorous testing methods to ensure the reliability and functionality of our platform. Utilising wallets generated by Anvil, a development tool that allows rapid and flexible blockchain interaction, we were able to conduct tests that simulate real-world transaction scenarios. This approach helped us to not only validate our platform's capabilities but also to refine and enhance its performance.

Solving Critical Challenges:

1. **Scalability:** Ethereum network congestion often leads to high gas fees and slow transaction processing times. By implementing a zk-rollup solution, VOCA is able to process transactions off-chain and batch them into a single on-chain transaction, drastically reducing costs and improving throughput.
2. **Security and Privacy:** Maintaining security without compromising on transaction speed is a significant challenge. VOCA leverages zero-knowledge proofs, ensuring that transaction validity is verified without exposing any sensitive user data.

Role of Anvil Wallets in Testing:

The use of Anvil for generating test wallets has been useful in our development and testing phases. By using predefined private keys to initialise wallets within our testing environment, developers can simulate user interactions more realistically, assessing how the system handles transactions both from a performance and security standpoint.

- Integration in Development: The **WalletService** module, as shown in the provided code snippet, initialises test wallets with Anvil-generated private keys. This allows our developers to quickly set up and interact with the Ethereum blockchain without the need for real funds.

```
// Define private keys for test wallets
export const testWalletKeys = {
  firstPrivateKey: "0x59c6995e998f97a5a0044966f0945389dc9e86dae88c7a8412f4603b6b78690d",
  secondPrivateKey: "0x5de4111afa1a4b94908f83103eb1f1706367c2e68ca870fc3fb9a804cdab365a"
};
```

- Test wallet Generation: In the dashboard, the test wallets can be generated with a single click, using buttons labelled "Generate Test Wallet 1" and "Generate Test Wallet 2." This functionality not only demonstrates the ease of wallet creation and management but also ensures that developers can safely interact with the platform without real-world repercussions. The picture below shows the first private key being used to make a wallet and generate the public key, and an instance of the rollup contract. The keys will be generated from seeding the user's eth wallet.

```
<button
  className={styles.walletButton}
  onClick={handleGenerateTestWallet1}
>
  Generate Test Wallet 1
</button>
<button
  className={styles.walletButton}
  onClick={handleGenerateTestWallet2}
>
  Generate Test Wallet 2
</button>
```

The screenshot shows a dark-themed dashboard titled "Generate Your Wallet". It displays a "Private Key" and a "Public Key", each followed by a "Copy key" link. Below the keys are three yellow buttons: "Generate Wallet", "Generate Test Wallet 1", and "Generate Test Wallet 2".

Generate Your Wallet

Private Key:
0x59c6995e998f97a5a0044966f0945389dc9e86dae88c7a8412f4603b6b78690d
[Copy key](#)

Public Key:
0xba5734d8f7091719471e717ed6b9df170dc70cc661ca05e688601ad984f068b0, 0xd67351e5f06073092499336ab0839ef8a521afd334e53807205fa2f08eec74f4
[Copy key](#)

[Generate Wallet](#)

[Generate Test Wallet 1](#)

[Generate Test Wallet 2](#)

- Debugging and Validation: Logging the initialization details helps in debugging and ensures that all components of the wallet (private key, public key components, and wallet address) are functioning as expected. This rigorous testing framework aids in preemptively identifying and resolving issues that could affect end-users.

```

Contract instance: DepositComponent.tsx:69
  ▶ Contract {interface: Interface, provider: JsonRpcProvider, signer: Wallet, callStatic: {...}, estimateGas: {...}, ...}
Account: 0x7420719dCa3f49b742aB008dE06f3601Ec656770 DepositComponent.tsx:70
Public Key: DepositComponent.tsx:71
  ▶ {x: '0xba5734d8f7091719471e7f7ed6b9df170dc70cc661ca05e688601ad984f068b0', y: '0xd67351e5f06073092499336ab0839ef8a521afd334e53807205fa2f08eec74f4'}

```

The incorporation of Anvil wallets into our testing strategy highlights overcoming the technical hurdles associated with blockchain technologies.

5. Results

The results section of this documentation provides a comprehensive overview of the outcomes from the implementation of the VOCA project. It focuses on user interactions such as wallet connection and fund deposit operations, and the backend functionality of the operator node, smart contracts and circuits.

Through detailed examples and results from live operations, we will showcase how the VOCA project meets its objectives and fulfils the needs of its users. By documenting these successes, we aim to provide a clear and measurable account of the project's utility.

Running the Operator Node:

The operator node can listen to the rollup smart contract and process them once they occur.

```

listenForDepositEvents() {
  try {
    this.contract.on('RequestDeposit', (pubKey, amount, tokenType) => {
      console.log(`Deposit received: pubKey[${pubKey}], amount[${amount}], tokenType[${tokenType}]`);
      this.handleDeposit({ pubKey, amount, tokenType }).catch(console.error);
    });
  } catch (error) {
    console.error('Error occurred while listening for deposit events:', error);
  }
}

```

Once a set of deposits comes in, the node then begins the processing, by hashing the deposit data and creating a subtree out of the deposits. We then create proofs of exclusion to ascertain if there is space within the tree for the batch of deposits

```

async processDepositsBatch() {
  const numLeaves = 2 ** this.BAL_DEPTH;

  const pendingDeposits = this.pendingDeposits.splice(0, 4);
  const pendingDepositsAccounts = [];
  const accounts = this.accountTree.accounts.slice(0, this.accountIdx);
  for (let i = 0; i < pendingDeposits.length; i++) {
    const { pubKey, amount, tokenType } = pendingDeposits[i];
    const acc = new Account(this.accountIdx++, pubKey[0], pubKey[1], Number(amount), 0, Number(tokenType));
    accounts[acc.index] = acc;
    pendingDepositsAccounts.push(acc);
  }
  const subtree = new AccountTree(pendingDepositsAccounts);
  const subtreeRoot = subtree.root;
  this.subtreeHashes.push(subtreeRoot);
  // after we have the proof, we should check the batch index to determine how many subtrees we need to fill
  // we should probably store each subtree hash in an array so we can progressively build the tree
  const subtreeProof = this.zeroCache.slice(1, this.BAL_DEPTH - Math.log2(4) + 1).reverse();
  if (this.batchIdx > 0) {
    for (let i = 0; i < this.batchIdx; i++) {
      subtreeProof[i] = this.subtreeHashes[i];
    }
  }
  const paddedAccounts = treeHelper.padArray(accounts, new Account(), numLeaves);
  console.log('root before processing deposits: ', this.accountTree.root.toString());
  console.log('contract root before: ', await depositTx.currentRoot());
  this.accountTree.accounts = paddedAccounts;
  this.accountTree.leafNodes = paddedAccounts.map(acc => acc.hashAccount());
  this.accountTree.innerNodes = this.accountTree.treeFromLeafNodes();
  this.accountTree.root = this.accountTree.innerNodes[0][0];
  console.log('accounts: ', this.accountTree.accounts);
  console.log('contract root1: ', await depositTx.currentRoot());
  const pos = treeHelper.idxToBinaryPos(this.batchIdx++, this.BAL_DEPTH - Math.log2(4));
  console.log('Processing deposits for batch:', this.batchIdx - 1, 'with pos:', pos, 'and subtreeProof:', subtreeProof);
  try {
    console.log('Processing deposits...');
    const txResponse = await depositTx.processDeposits(2, pos, subtreeProof, {gasLimit: 1000000});
    const txReceipt = await txResponse.wait(); // waits for the transaction to be mined

    if (txReceipt.status === 1) {
      console.log('Transaction successful:', txResponse.hash); // Correctly log the transaction hash from the response
      console.log('root from contract equals root calculated? : ', this.accountTree.root, await depositTx.currentRoot());
    } else {
      console.log('Transaction failed without throwing an error, receipt:', txReceipt);
    }
  } catch (error) {
    console.error('Error processing deposits:', error);
  }
}
}

```

You, last week • #17 implement deposit workflow. Closes 17.

Once this is done, we then post the data along with the proof to the smart contract, which then updates the state on the chain.

Once these deposits are processed, users can then start transacting with other users on the network. The operator receives these transactions and batches them.

```

processNextBatch() {
  const transactions = this.transactionPool.getNextBatch(this.batchSize);

  console.log('transactions: ', transactions);

  const txTree = new TxTree(transactions);
  const stateTransition = this.accountTree.processTxArray(txTree);

  const circuitInput = getCircuitInput(stateTransition);
  this.submitProofGeneration(circuitInput);

  this.onBatchProcessed(transactions, circuitInput);
  this.transactionPool.confirmProcessedBatch(transactions);
}

```

Once batched, the operator begins to process the transactions and alter the state of the tree. Upon completion, the node takes the inputs and begins the proof generation process.

```

async submitProofGeneration(circuitInput) {
  console.log("Submitting for proof generation:", circuitInput);
  try {
    const { proof, publicSignals } = await snarkjs.groth16.fullProve(circuitInput, path.join(__dirname, 'batch.wasm'), this.zkey);
    console.log("Public Signals:", publicSignals);
    console.log("Proof:", proof);

    const verificationKeyJson = await fs.readFile(this.vkey, { encoding: 'utf8' });
    const verificationKey = JSON.parse(verificationKeyJson); // Parse the JSON string into an object.

    const isValid = await snarkjs.groth16.verify(verificationKey, publicSignals, proof);

    if (!isValid) {
      throw new Error("Proof generation failed");
    } else {
      // submit proof to the contract
      console.log("Proof generation successful");
      console.log('local root: ', this.accountTree.root.toString());
      console.log('contract root: ', await rollupContract.currentRoot());
      const update = await rollupContract.updateState(
        [
          proof.pi_a[0], proof.pi_a[1]
        ],
        [
          [proof.pi_b[0][1], proof.pi_b[0][0]],
          [proof.pi_b[1][1], proof.pi_b[1][0]],
        ],
        [
          proof.pi_c[0], proof.pi_c[1]
        ],
        publicSignals
      );

      const receipt = await update.wait();
      if (receipt.status === 1) {
        console.log('Update transaction confirmed.');


const newRoot = await rollupContract.currentRoot();



console.log('New contract root:', newRoot.toString())


```

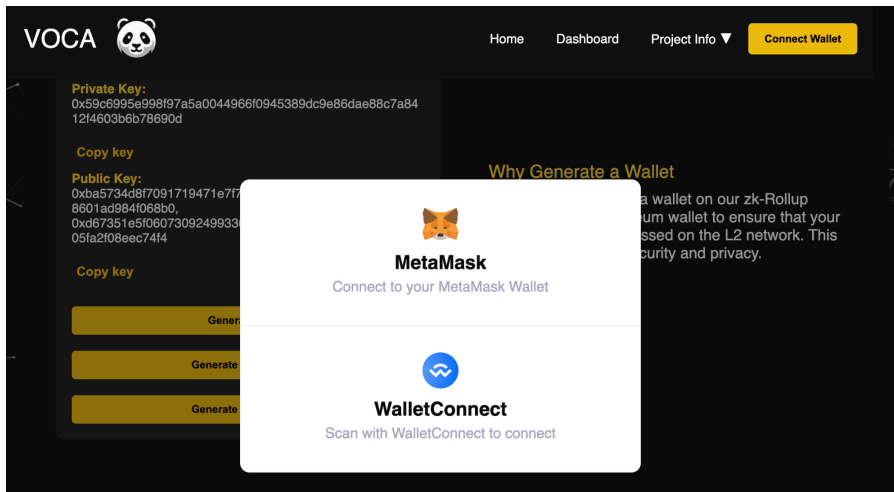
Once the proof is generated, we then call the smart contract to update the state of the network on the chain. If the proof is valid and the call was successful, then we will update the state locally.

Unfortunately, due to time constraints and the challenging nature of this project, we were unable to complete the withdrawal process for users, and our operator node fails to process more than two batches of deposits.

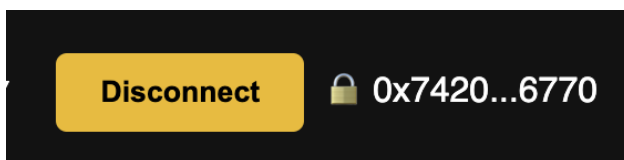
Connecting a users ethereum wallet:

Screenshots and Details:

- **Wallet Connection Interface:** The screenshots demonstrate the user interface where users initiate the wallet connection. This includes the "Connect Wallet" button prominently displayed, guiding users through the process.
- **Wallet Provider Selection:** they also capture the modal window presented to the user, listing available wallet providers such as MetaMask and WalletConnect. This step is crucial as it offers users the flexibility to choose their preferred method of connection.



- **Successful Connection Confirmation:** Additional screenshots show the application's state post-connection. These images highlight the UI update that displays the user's wallet address, confirming a successful linkage between the user's wallet and the VOCA platform, with a disconnect option.



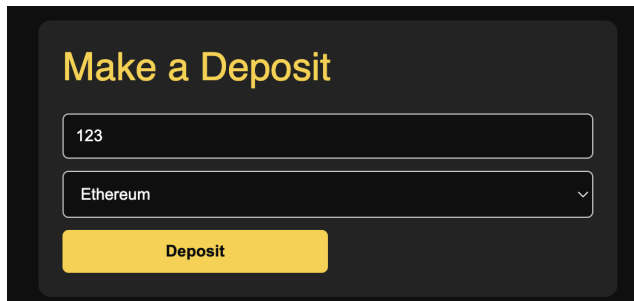
User Feedback and System Performance:

- **User Experience:** Users have reported a smooth and intuitive connection process, appreciating the ease of interaction and the quick response of the system.
- **Performance Metrics:** The connection setup has been tested under various network conditions to ensure robust performance, including rapid response times and high reliability rates.

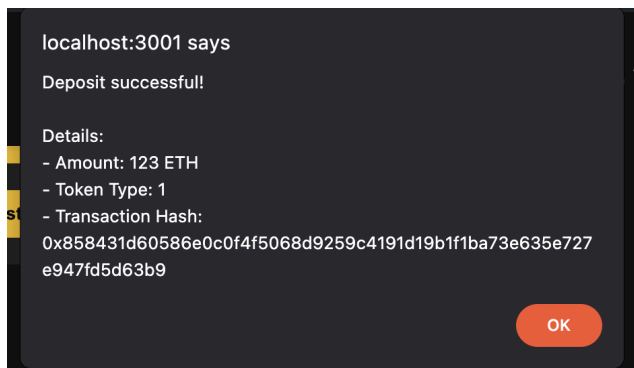
Depositing Funds:

Screenshots and Details:

- **Deposit Interface:** Screenshots display the deposit section accessible through the platform's dashboard. Users are presented with a straightforward form where they can specify the amount and type of currency or token they wish to deposit. Along with the initial deposit of the amount, token type and deposit button



- **Successful Transaction Feedback:** Screenshots of the alert window provide users with immediate feedback upon the successful submission and processing of their deposit. This alert confirms the transaction details, including the amount deposited and the transaction hash, providing transparency and reassurance to the user.



- **Backend Transaction Processing:** Additional visual evidence from the backend logs or administrative panels demonstrates that the transaction has been processed correctly. This includes confirmation of the transaction details, such as the recipient's address, amount, and timestamp, which are crucial for maintaining an auditable trail of operations.

```
Transaction: 0x858431d60586e0c0f4f5
068d9259c4191d19b1f1ba73e635e727e947fd5
d63b9
Gas used: 170339

Block Number: 19
Block Hash: 0x0c94a5ebfcd4ce8745408
aae210efc7e0b9d7baffc029a7ca30f8baaa8bb
2201
Block Time: "Sun, 21 Apr 2024 14:26
:01 +0000"

eth_chainId
eth_getTransactionReceipt
```

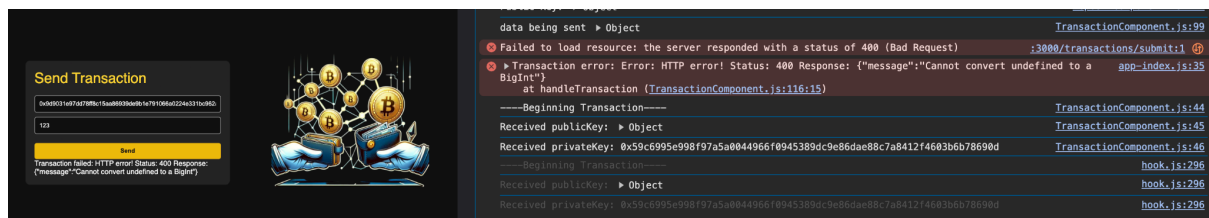
User Feedback and System Performance:

- **User Experience:** Users have positively highlighted the ease and clarity of the deposit process. The intuitive UI and clear feedback mechanisms help reduce confusion and build confidence in the platform's capabilities.

6. Future Work

Enhancements to TransactionComponent on the frontend

One notable error that users face is the "Cannot convert undefined to a BigInt" error, which indicates a problem in the data transmission or type handling between the frontend and the backend. This issue is critical as it prevents the successful submission of transactions via the frontend, forcing transactions to be manually carried out on the backend



Planned Improvements:

1. **Robust Error Handling and Validation:**
 - **Frontend Enhancements:** Improve input validation to ensure that all data sent to the backend conforms to expected formats, particularly focusing on numerical values that must be converted to `BigInt`. This involves enhancing the logic within the `TransactionComponent` to perform comprehensive checks before initiating a transaction.
2. **Type Safety Enhancements:**
 - **Data Flow Review:** Conduct a thorough review of how data flows from the user input through to the POST call to ensure that data integrity is maintained at every step. This should include tracing and logging enhancements to make the debugging process more straightforward.
3. **User Feedback Improvements:**
 - **Progress Indicators:** Implement progress indicators and more detailed transaction status updates to keep the user informed about each step of the transaction process, from initiation to completion or failure