1. Abstract
2. Background / Problem
   1. Description of Problem
      1. Summary
         1. Under the FDA, there is very strict guidance around complaint handling for medical devices as outlined by 21 CFR 820.198. Medical device complaints must be processed in a timely manner, and with an auditable trail, to ensure that all events are investigated and can be reported to the FDA if needed. Most companies have a quality management system (QMS) which is structured to support compliance for medical device complaint reporting. These complaints are also consumed by other portions of the business outside of the quality compliance groups, such as
            1. \* Integration in a product backlog as a bug or feature enhancement request
            2. \* For service teams to have up-to-date information and tracking on customer issues and for scheduling service
            3. \* For tagging to inventory systems to support regulatory restrictions or awareness of product issues
            4. \* Integration with the design history file (DHF) of a product
            5. \* Customer-facing personnel may log observations or information outside of complaints which also needs to be tracked
         2. In many instances, the software programs used in the applications listed above likely do not easily interface with one another. This causes issues where the information from a customer complaint is not consistently distributed, updated, and tracked across all different domains, resulting in disrupted information flow.
         3. Additionally, these traditional database programs used for defect tracking are expensive and proprietary. This makes developing interfaces between them and all of the other programs used across the business difficult if not impossible as it requires an agreed interface between many disparate programs. It also serves as a cost barrier for medical device startups.
      2. Requirements  
         To ensure the dApp meets the needs of FDA 21 CFR 820.198 which outlines the regulations for medical device complaint tracking, the following requirements have been outline.
         1. \* All complaints must be stored indefinitely and are subject to audit.
         2. \* All complaint initial filings, addendums, and closures shall be be documented.
         3. \* Complaints in the system shall contain the following information at minimum:
         4. \* Summary (Title)
         5. \* Description
         6. \* Site of complaint
         7. \* Severity
         8. \* Corrective action (if applicable)
         9. \* Product
         10. \* The system shall allow linking of complaints, addendums, and closures.
         11. \* The system shall allow retrieval of a complaint given an ID.
         12. \* The system shall allow retrieval of all linked complaints given a complaint ID.
         13. \* Complaint records should be protected from modification after the initial filing.
         14. \* The system shall be designed such that the complaint records are stored redundantly to prevent data loss.
   2. Current Solutions
   3. Current Opportunities of Ethereum dApp’s
      1. Project Objective  
         The objective of this project is to provide a proof-of-concept for an open-sourced dApp and set of API's for common consuming programs which is designed to meet the requirements for medical device complaint handling by the FDA. This foundation of well-defined documents and proof-of-concept dApp will provide an artifact to drive futher open-sourced work and discussion on innovations in the medical device community, and an opportunity for a lightning talk topic at a future medical device conference.
      2. Goal  
         This project proposes an open-source, FDA compliant, secure, medical device complaint management system powered by the concept of the decentralized ledger as a decentralized application (dApp).
3. dApp Implementation
   1. Architecture  
      A picture containing diagram

      Description automatically generated
      1. Developed in the Truffle Development Suite
      2. The dApp has three primary components - the Ethereum blockchain, the web3 interface, and the client side of the application.
         1. The Ethereum Blockchain makes up the "back end" of the dApp.
            1. In this proof-of-concept a local Ethereum blockchain is generated and hosted by the [Ganache](https://trufflesuite.com/ganache/) dev application which is run locally.
            2. The contract was written in the [Solidity](https://soliditylang.org/\_) language which is "a statically-typed curly-braces programming language designed for developing smart contracts that run on Ethereum." It was written and deployed using the [Truffle](https://trufflesuite.com/\_) framework which simplifies and expedites development.
            3. Contract unit testing was done in [JavaScript](https://www.javascript.com/) using the [Mocha](https://mochajs.org/) testing framework. Unit testing of ethereum contract code is critical to ensure proper functionality of the contract prior to deployment, as it cannot easily be changed post-deployment without migration, which, for large networks can be logistically complex.
         2. The web3 interface connects the backend of the blockchain to the front client-facing end.
            1. Both [Google Chrome](https://www.google.com/chrome/) and the [Google Chrome Extension](https://sites.google.com/coinsloginn.com/metamaskcromeextension/home?pli=1) for [MetaMask](https://metamask.io/) are used to facilitate connecting the user to the locally hosted blockchain.
            2. The actual code to connect the browser tools to both the local blockchain and the user was done using [Web3.js](https://web3js.org/#/) which is a [JavaScript](https://www.javascript.com/) library for building on Ethereum.
         3. The client side of the dApp is what the user actually interacts with. It is composed of the user themselves and the front-end of the dApp which was written using HTML5 and CSS. Graphical user interface

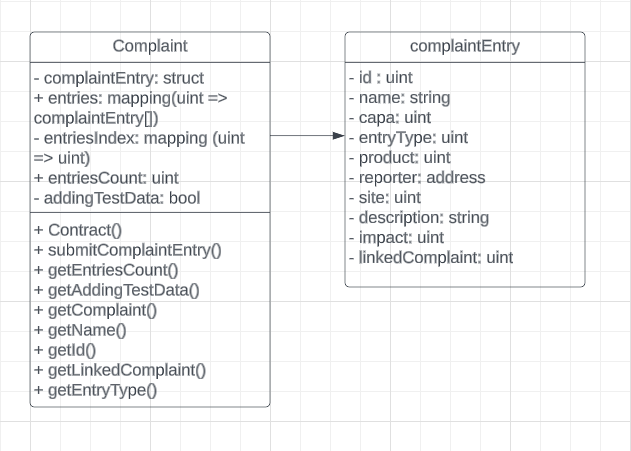
            Description automatically generated with medium confidence
            1. This proof-of-concept dApp has been built with five pre-existing block entries on which the functionality can be easily tested.
            2. When the dApp is launched, the contract will automatically be populated with these five example complaints.  
               Table, Excel

               Description automatically generated
            3. Current Complaints  
               On the left-hand side you can see that the example data populated by the contract creates five blocks, each representing individual complaints. Notice that complaints three and four are linked to complaint two, and complaint five is linked to complaint one.  
               Table

               Description automatically generated
            4. Print Linked Complaints  
               When the dApp is launched, you can request the linked complaints be returned as shown below.  
               Text

               Description automatically generated with low confidence
            5. Submitting a New Complaint  
               On the right hand side of the screen is a form to submit a new complaint. If you enter information and hit submit, the contract will check if the entry is valid - i.e. valid entry type, impact type, and linked complaint (if applicable) - before adding it to the chain. If you enter the following dummy data...  
               Graphical user interface, text, application

               Description automatically generated  
               You'll see that your MetaMask extension will prompt you for a transaction confirmation. Click 'Submit' to allow the transaction to complete.  
               Graphical user interface, text, application

               Description automatically generated  
               Upon completion the screen will refresh and you'll now see Complaint 6 has been added to the list of current complaints, with block id 6 and linked complaint 1.
   2. Code Structure
      1. Contract Structure  
           
         The contract, named \*Complaint\*, is made up of a a series of complaint entries and containers and objects to store and keep track of those entries.
         1. \*complaintEntry\* - a custom struct that models the characteristics of the complaints, where each complaint will be its own block in the chain
         2. \*entries\* a mapping of arrays of complaintEntries to uint keys to keep track of the complaints
         3. \*entriesIndex\* - a mapping of uints to uints to keep track of an entry's position on the block so that it may easily be retrieved
         4. \*entriesCount\* - a numeric count of the number of blocks in the chain
         5. Additionally there is a \*submitEvent\* event object used to trigger refreshes on the application.
      2. Complaint Entries  
         The contract entries are structured as shown above and detailed below. While only the Name, ID, and LinkedComplaint characteristics are relevant for user interaction, the other characteristics have been created to demonstrate the use case for tracking medical complaints.Additionally, some of the "extra" characteristics are used to ensure entries are valid.
         1. - \*id\* - a unique identifier for the complaint which also matches its block ID on the chain.
         2. - \*name\* - the user-given name of the complaint.
         3. - \*capa\* - a numeric identifier to link a complaint to a corrective action plan (capa).
         4. - \*entryType\* - a numeric enumeration that indicates the type of entry, where 0 = complaint, 1 = addition, 2 = closure.
         5. - \*product\* - a numeric identifier to map a complaint to a product sku
         6. - \*reporter\* - the ethereum address used to submit the complaint.
         7. - \*site\* - a numeric identifier to identify the customer site for the complaint.
         8. - \*description\* - a short description of the complaint.
         9. - \*impact\* - a numeric enumeration that indicates the impact of the complaint where 0 = observation, 1 = low, 2 = moderate, 3 = high, 4 = SAFETY.
         10. - \*linkedComplaint\* - a numeric identifier which can be used to link multiple complaints together, 0 indicates no link.
      3. Functions  
         The contract functions are shown in the image above. They are used to instantiate the contract, populate dummy data, add complaints to the chain, and retrieve information.
         1. - \*Complaint()\* - this is the constructor which creates the contract, it also populates the contract with five sample complaints without emitting any events that would cause the application to refresh.
         2. - \*submitComplaintEntry()\* - takes in all parameters of a complaint with the exception of id and reporter to submit a new complaint. The id will be given by the contract and will be the number of the next available block. The reporter address will be the ethereum address associated with the dApp instance that is submitting the complaint.
         3. - \*getEntriesCount()\* - returns the number of blocks on the chain.
         4. - \*getAddingTestData()\* - returns the value of the boolean \*addingTestData\* which is used to prevent refresh-triggering events when populating the contract with test data.
         5. - \*getComplaint()\* - given a block id (i.e. complaint id) it returns a complaintEntry object.
         6. - \*getName()\* - given a block id (i.e. complaint id) it returns the name assocaited with that complaint entry.
         7. - \*getId()\* - given a block id (i.e. complaint id) it returns the id associated with that complaint entry.
         8. - \*getLinkedComplaint()\* - given a block id (i.e. complaint id) it returns the id of a linked complaint for that complaint entry if it exists.
         9. - \*getEntryType()\* - given a block id (i.e. complaint id) it returns the entry type of the complaint entry.
      4. Testing  
         A test for all functions in the contract is in the test subdirectory of the dApp. It uses both the existing example data as well as some new data. Recall the original example data populates when you run the contract. Ten different test cases.
         1. Correct Example Data is Generated
         2. Correct parameters for test data
         3. Emit events don’t occur until after test data populates
         4. A new complaint can be submitted
         5. A valid entry type is required to submit a complaint
         6. A valid impact type is required to submit a complaint
         7. A valid linked complaint is required to submit a complaint
         8. A complaint name can be retrieved given an index
         9. A complaint id can be retrieved given an index
         10. A complaint’s linked ID can be retrieved given an index
4. Future Work
   1. Common API development
   2. Add more robust boundary testing
   3. Update front-end layout
   4. Implement enums for entry type and impact type
   5. Implement drop down menus in submission form
   6. Validate product id against external server
   7. Validate site id against external server
   8. Add function to retrieve previous transaction by hash
   9. Add reviewer functionality to complaints
   10. Implement hash / optimization of linked lookup
   11. Improve linking function to identify multiple links
   12. Add ability to link multiple complaints
   13. Improve print function to retrieved all chained complaints
   14. Implement user-facing error messages
   15. Add from end testing for UI
5. Conclusion