Meta Scan Final Design Report

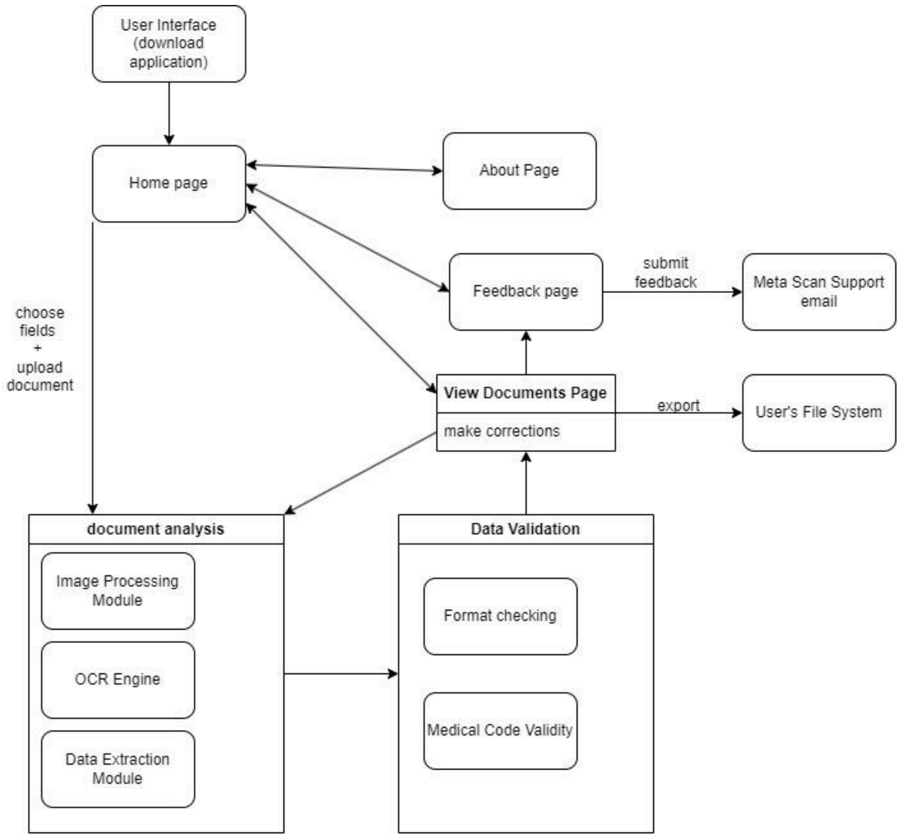
By: Jessica Luu, Arman Seth, William Struve, Ali Naqvi

Design

Logical Organization

MetaScan is separated into two main sections, the front end and the back end.

The front end is in charge of webpages structures and design using HTML and CSS. The webpages are found in WindowsStuff > Templates and there are a total of four web pages. The webpages are styled using CSS, and the style.css file can be found within WindowsStuff > Static.



For the backend, the majority of the functionality for the web pages are kept within one giant file, app.py, which you can navigate by going into the WindowsStuff folder. We used Flask in order to manage the backend as we had already begun writing functions in python and we had some knowledge of using Flask as well. To connect the front end with the back end, we use JavaScript event listeners to keep track of dynamic changes to the web pages. These dynamic changes are used by the back end to perform operations such as switching web pages and uploading UB-04 forms.

Languages
Languages used for MetaScan include Python, JavaScript, HTML, and CSS. Python was used in the backend using the Flask framework. We chose to use Flask as it is lightweight and we had already started writing some of the scripts in Python. HTML was used for structuring the webpages in the front end, creating the main sections and key elements of the pages. CSS was used for styling the webpages, making them easy to navigate and interact with. JavaScript was used to connect the front end to the backend. Using event listeners, we are able to utilize real time updates to buttons and checkboxes

Libraries
Libraries in python that were used for this project include PyTesseract, SMTPlib, and Flask. Pytesseract was used to extract text from the submitted form images while also verifying that the correct form was submitted. Smtplib was used in the feedback section to allow us to connect with the SMTP server and to send emails to the MetaScan support email. Flask was used to build the web application and it handled the routing, requests and responses, and server-side functionalities.

System Usage and Operation
In order to operate the system, the Client would need to install git and python on their machine. They would clone the repository using “git clone https://github.com/CS-179K/Meta-Scan.git” in order to download the code onto their machine. They would also need to install Pytest, Pytesseract, and flask in order to run the program through their respective “pip install” commands. The client can then navigate to the WindowsStuff folder by using the command “cd WindowsStuff.” The title “WindowsStuff” is a holdover of the part of the project where we sought to make it work on different operating systems but it is the final version that should work on everything. To run the program, would then run the command “python app.py” and when the program starts, a link would appear in the terminal to the local host and, when they click on it, they have access to the program’s webpage.

User Interaction
In order for a user to use the system after the client has set it up, they would be first met with the home page. On the home page, the user has the ability to navigate to all other web pages (the about page, the help page, the feedback page, and the documents page), choose the fields they want extracted from their UB-04 form, and submit a UB-04 form. To choose a field the user can check off the category they would like to receive information from. To upload a UB-04 form, the user can click on the upload button, where they can choose a file from their computer’s file system. When they upload a file, if that form is accepted, they will be taken to the documents page, where they are able to see all the information extracted from their form. If the form is rejected, they will be notified and will be kept on the home page. At the bottom of the documents page, the user can choose to upload more forms, export the current extracted info as a json file, or edit the fields that were extracted improperly. On the about page, the user can read a brief description of MetaScan and what MetaScan can do. On the help page, the user can read a tutorial that helps them use MetaScan for the first time. On the feedback page, there is a drop down for the user to select their feedback type and a text box for the user to write their

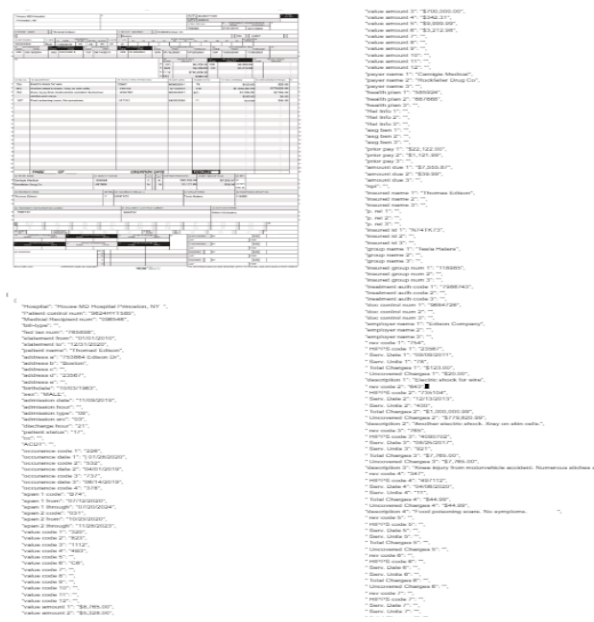
feedback. This feedback can be sent by pressing the submit button and if the feedback goes through, they will receive a confirmation message.

Evaluation

1. Testing the effectiveness of OCR and extraction

   Overview: We can test the effectiveness of this portion of the code by constructing sample ub04 forms similar to how they're made in the industry and then uploading them through the website. The ultimate goal of the project is a final downloadable json so we can compare the known values of the field in the image to the values returned to see what was successful.

   Result: To best show how well this function was implemented, I will show a picture of a sample input and its corresponding json although the testing for this was done with many runs of many different files.



```
[
  {
    "Hospital": "House MD Hospital Princeton, NY ",
    "Patient control num": "9824HY1585",
    "Medical Recipient num": "098548",
    "bill-type": "",
    "fed tax num": "785898",
    "statement from": "01/01/2010",
    "statement to": "12/31/2020",
    "patient name": "Thomad Edison",
    "address a": "753864 Edison Dr",
    "address b": "Boston",
    "address c": "",
    "address d": "23567",
    "address e": "",
    "birthdate": "10/03/1983",
    "sex": "MALE",
    "admission date": "11/09/2019",
    "admission hour": "",
    "admission type": "09",
    "admission src": "03",
    "discharge hour": "21",
    "patient status": "17",
    "cc": "",
    "ACDT": "",
    "occurance code 1": "226",
    "occurance date 1": "j 01/26/2020",
    "occurance code 2": "532",
    "occurance date 2": "04/01/2019",
    "occurance code 3": "737",
    "occurance date 3": "08/14/2019",
    "occurance code 4": "378",
    "span 1 code": "874",
    "span 1 from": "07/12/2020",
    "span 1 through": "07/20/2024",
    "span 2 code": "031",
    "span 2 from": "10/23/2020",
    "span 2 through": "11/28/2023",
    "value code 1": "320",
    "value code 2": "823",
    "value code 3": "1112",
    "value code 4": "483",
    "value code 5": "",
    "value code 6": "C8",
    "value code 7": "",
    "value code 8": "",
    "value code 9": "",
    "value code 10": "",
    "value code 11": "",
    "value code 12": "",
    "value amount 1": "$8,765.00",
    "value amount 2": "$5,328.00",
    "value amount 3": "$700,000.00",
    "value amount 4": "$342.31",
    "value amount 5": "$9,999.99",
    "value amount 6": "$3,212.98",
    "value amount 7": "",
    "value amount 8": "",
    "value amount 9": "",
    "value amount 10": "",
    "value amount 11": "",
    "value amount 12": "",
    "payer name 1": "Carnegie Medical",
    "payer name 2": "Rockfeller Drug Co",
    "payer name 3": "",
    "health plan 1": "385924",
    "health plan 2": "887888",
    "health plan 3": "",
    "Rel Info 1": "",
    "Rel Info 2": "",
    "Rel Info 3": "",
    "asg ben 1": "",
    "asg ben 2": "",
    "asg ben 3": "",
    "prior pay 1": "$22,122.00",
    "prior pay 2": "$1,121.99",
    "prior pay 3": "",
    "amount due 1": "$7,555.87",
    "amount due 2": "$39.99",
    "amount due 3": "",
    "npi": "",
    "insured name 1": "Thomas Edison",
    "insured name 2": "",
    "insured name 3": "",
    "p. rel 1": "",
    "p. rel 2": "",
    "p. rel 3": "",
    "insured id 1": "N74TK73",
    "insured id 2": "",
    "insured id 3": "",
    "group name 1": "Tesla Haters",
    "group name 2": "",
    "group name 3": "",
    "insured group num 1": "118985",
    "insured group num 2": "",
    "insured group num 3": "",
    "treatment auth code 1": "7988743",
    "treatment auth code 2": "",
    "treatment auth code 3": "",
    "doc control num 1": "9884726",
    "doc control num 2": "",
    "doc control num 3": "",
    "employer name 1": "Edison Company",
    "employer name 2": "",
    "employer name 3": "",
    " rev code 1": "754",
    " HIPPS code 1": "23567",
    " Serv. Date 1": "09/09/2011",
    " Serv. Units 1": "78",
    " Total Charges 1": "$123.00",
    " Uncovered Charges 1": "$20.00",
    "description 1": "Electric shock for wire",
    " rev code 2": "643",
    " HIPPS code 2": "735104",
    " Serv. Date 2": "12/13/2013",
    " Serv. Units 2": "430",
    " Total Charges 2": "$1,000,000.99",
    " Uncovered Charges 2": "$779,820.99",
    "description 2": "Another electric shock. Xray on skin cells.",
    " rev code 3": "765",
    " HIPPS code 3": "4090702",
    " Serv. Date 3": "08/25/2017",
    " Serv. Units 3": "921",
    " Total Charges 3": "$7,765.00",
    " Uncovered Charges 3": "$7,765.00",
    "description 3": "Knee injury from motorvehicle accident. Numerous stitches and xrays.",
    " rev code 4": "347",
    " HIPPS code 4": "497112",
    " Serv. Date 4": "04/06/2020",
    " Serv. Units 4": "11",
    " Total Charges 4": "$44.99",
    " Uncovered Charges 4": "$44.99",
    "description 4": "Food poisoning scare. No symptoms.                ",
    " rev code 5": "",
    " HIPPS code 5": "",
    " Serv. Date 5": "",
    " Serv. Units 5": "",
    " Total Charges 5": "",
    " Uncovered Charges 5": "",
    " rev code 6": "",
    " HIPPS code 6": "",
    " Serv. Date 6": "",
    " Serv. Units 6": "",
    " Total Charges 6": "",
    " Uncovered Charges 6": "",
    " rev code 7": "",
    " HIPPS code 7": "",
    " Serv. Date 7": "",
    " Serv. Units 7": "",
```

Here is a screenshot of a ub04 form and its corresponding json data placed next to it. The final json files contain over 200 fields so it would be impossible to show every single one in this report alongside its values but reading over this result the highlights are:

- Each uploaded document returns 234 keys or properties
- How many are actually read is determined by what the user filled in on their form as not everything is required or requested by the user
- For this test:

        - 95 fields of the form were filled and 83 fields in the json had data. So the tesseract failed to read 12 inputs
            - A pattern among the unread fields was that they were all small fields with potentially one or two digit inputs. Most of these kinds of inputs were successfully captured but some were not.
        - Of the 83 captures, none of them had any mistakes or misread characters.

Similar success metrics were common across all sample ub04 forms. We noticed misread values about once every few forms.

Hard Truths: The nature of optical character recognition (OCR) is that it doesn't communicate with the uploaded data so it has to independently read many hundreds of inputs. As a result, the results for any document are not guaranteed to be one-hundred percent accurate. Some fields may not be read but we can reliably say that misread characters are rare. As a result, it may be best for a user to double check the results of the software with their documents for missed values which is still a faster process than manually constructing their own data. Overall, however, it is a success that hundreds of fields can be read successfully in a single run with a few errors.

What to Do Next: As stated it would be wise as a user to double check for errors and discrepancies. The user can then continue uploading additional documents until they are ready to download the complete json.

2. Testing the effectiveness of data validation

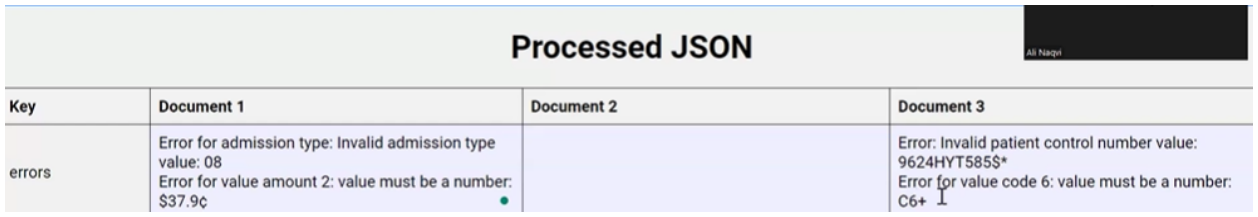
Overview: The main way to test our error detection was to fill in fields in the form with inputs that should be correct and inputs that should be caught as errors and testing if that is occurring with uploaded files. We have a folder for unit tests that checks to make sure that the validate functions are working properly on inputs generated by us:

```
#--Testing directly from ../WindowsStuff/app.py
#Date Format Regex Validation
@pytest.mark.parametrize("date_string, expected_result", [
    ("12-31-2023", True),
    ("01-01-2024", True),
    ("13-01-2024", False),  # Invalid month
    ("11-34-2024", False),  # Invalid day
    ("2024-08-26", False),  # Incorrect format
    ("08-26", False),  # Missing year
])
```

and then using them on inputs generated by the program. Three tests were conducted for three forms:

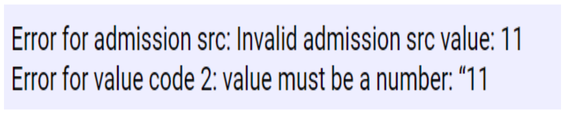
- Form 1: A field that is supposed to have a finite set of inputs is given a non-recognized input.
- Form 2: A form that should be perfect and return no error.
- Form 3: The same form but with three errors added through special characters.
- Form 4: A mix of special character errors and unacceptable input errors.

Result:



Processed JSON

Key	Document 1	Document 2	Document 3
errors	Error for admission type: Invalid admission type value: 08 Error for value amount 2: value must be a number: $37.9¢		Error: Invalid patient control number value: 9624HYT585$* Error for value code 6: value must be a number: C6+

Document 4: 

Error for admission src: Invalid admission src value: 11
Error for value code 2: value must be a number: “11

- Test 1: The intentional error was in the property “admission type” and returned false because “8” is not an accepted code. An unintended error was “value amount 2” with the value “$37.99” being read as “37.9¢”.
- Test 2: No errors as expected.
- Test 3: Two of the incorrect inputs were caught because they have unacceptable characters. A third input was not detected but that was because the character “ ‘ “ was appended to the front but not detected.
- Test 4: All incorrect values successfully returned as errors.

  Hard Truths: The error validation worked perfectly when it detected the inputs in fields with errors and did not return any false errors. It only missed errors in fields that were not properly read. Fortunately, it is very rare for OCR to make a mistake reading a character like it did for input #3 for the very small “ ‘ “ character. Equally fortunate is that it is possible for the program to detect its own errors and then allow the user to fix them if the misread input causes its own error. A limitation of this is that some misread values might still be acceptable and not be readily noticed by the program or user.

  What to Do Next: The user does not have to manually fix the errors (errors because the values they wrote are unacceptable) in their document that were detected but they have the ability to. Once they are satisfied, they can download the data.