LaserEye: Engineering RFC

Authors

- Kristen Hagan - kristen.hagan@duke.edu [back-end]
- David Li - jianwei.d.li@duke.edu [front-end]
- Jessica Loo - jessica.loo@duke.edu [image processing]

Reviewers

- Suyash Kumar - suyash.kumar@duke.edu
- Mark Palmeri - mark.palmeri@duke.edu

Table of Contents

Abstract

Briefly describe what this service/feature does and why. You may wish to highlight certain implementation details here.

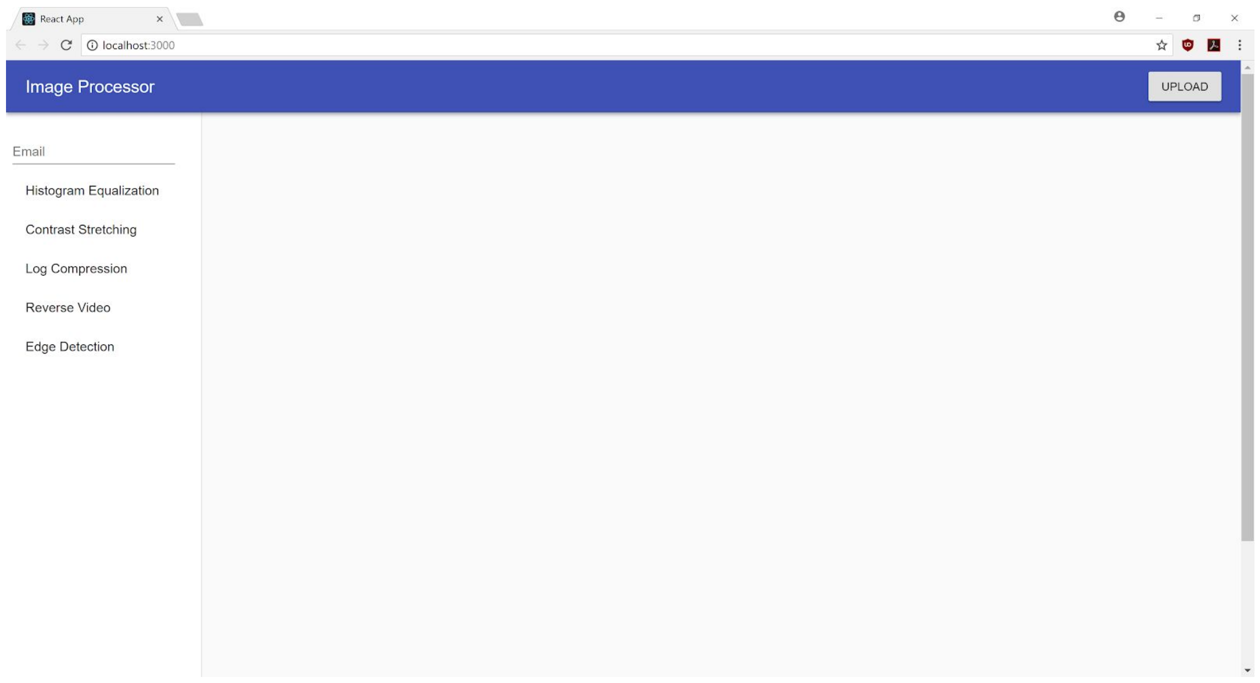
This is an online photo-editing service which will allow users to upload images (as a single file, a list of files, or a ZIP file) to a web-server, process them with a selection of image processing tools, and download the processed images. The web application (user interface) will be implemented in JavaScript using the React library. It will transfer the uploaded images to a RESTful web service implemented in Python using the Flask framework for image processing. The uploaded and processed images will be stored on the local disk, with associated metadata stored in a MongoDB database. The image processing will be implemented in Python, using image processing libraries such as scikit-image.

Background

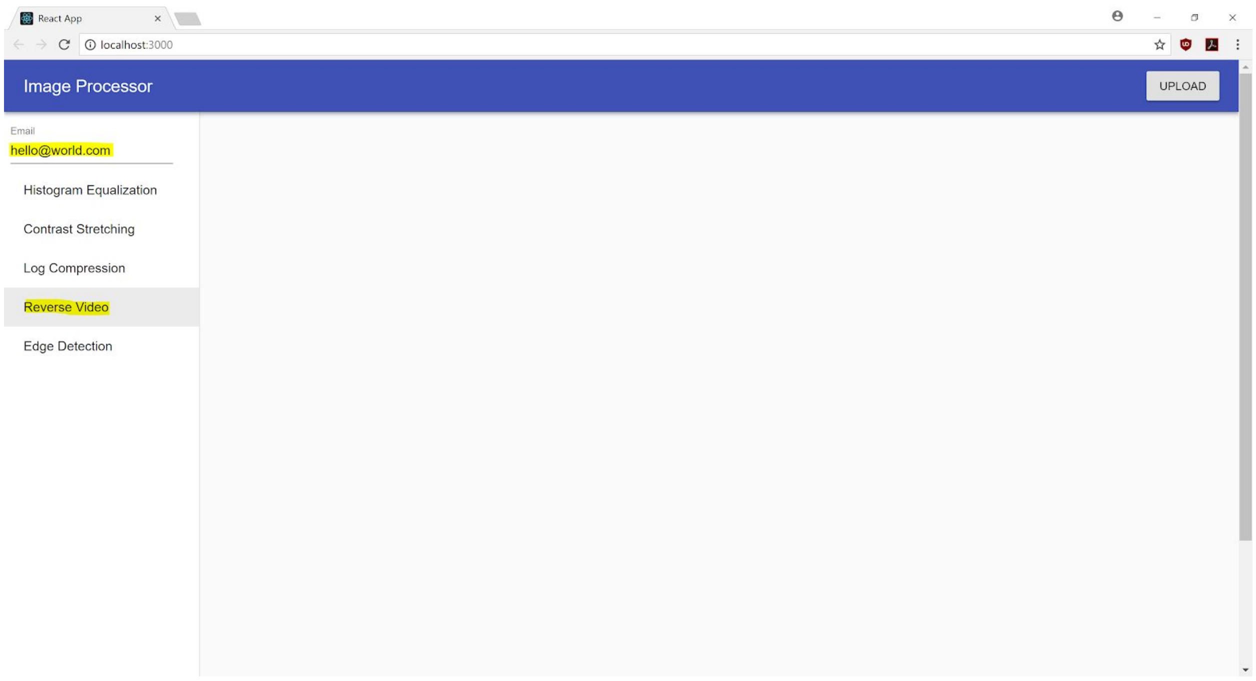
If there is a *Feature Narrative* or *Product Problem Statement*, link it here. Describe the motivations and goals of this product, along with any backstory that'll help RFC readers quickly understand relevant context.

The motivation and goals of this product is to provide a convenient basic image processing service, which does not require downloading and installing external software on the user's local machine.

Designs (front-end)

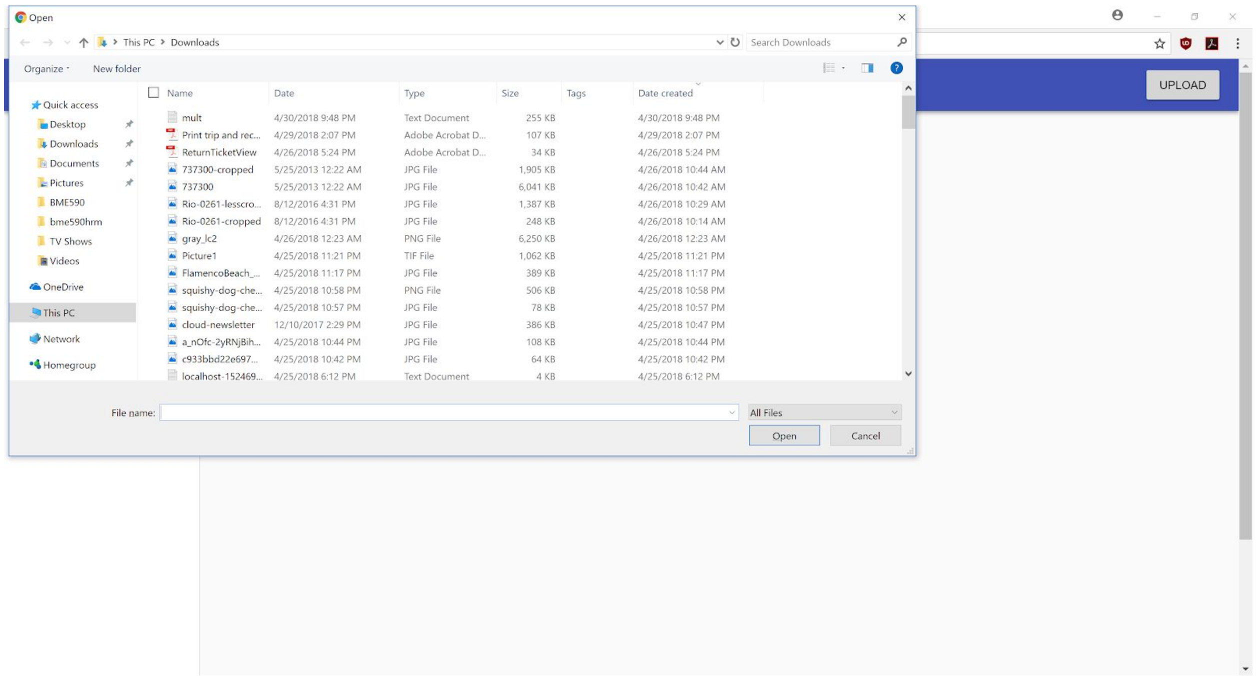


Default state upon starting the React app



Email: hello@world.com

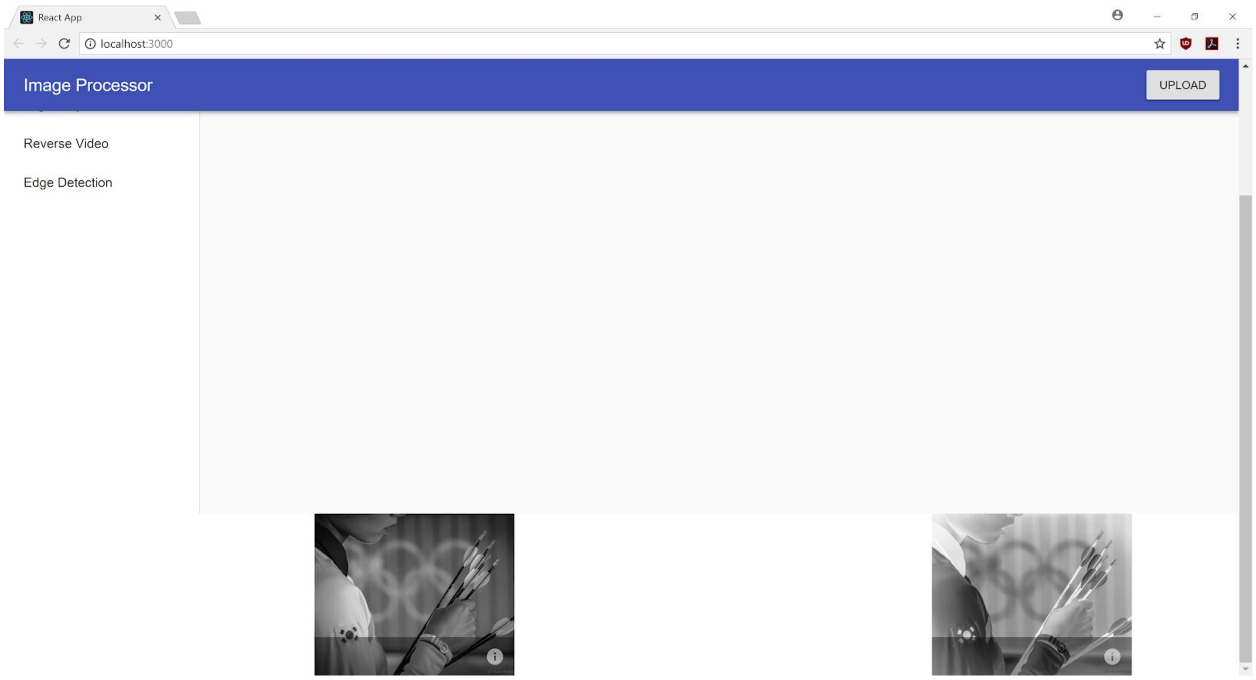
Text field with filled user email and the reverse video processing command selected.



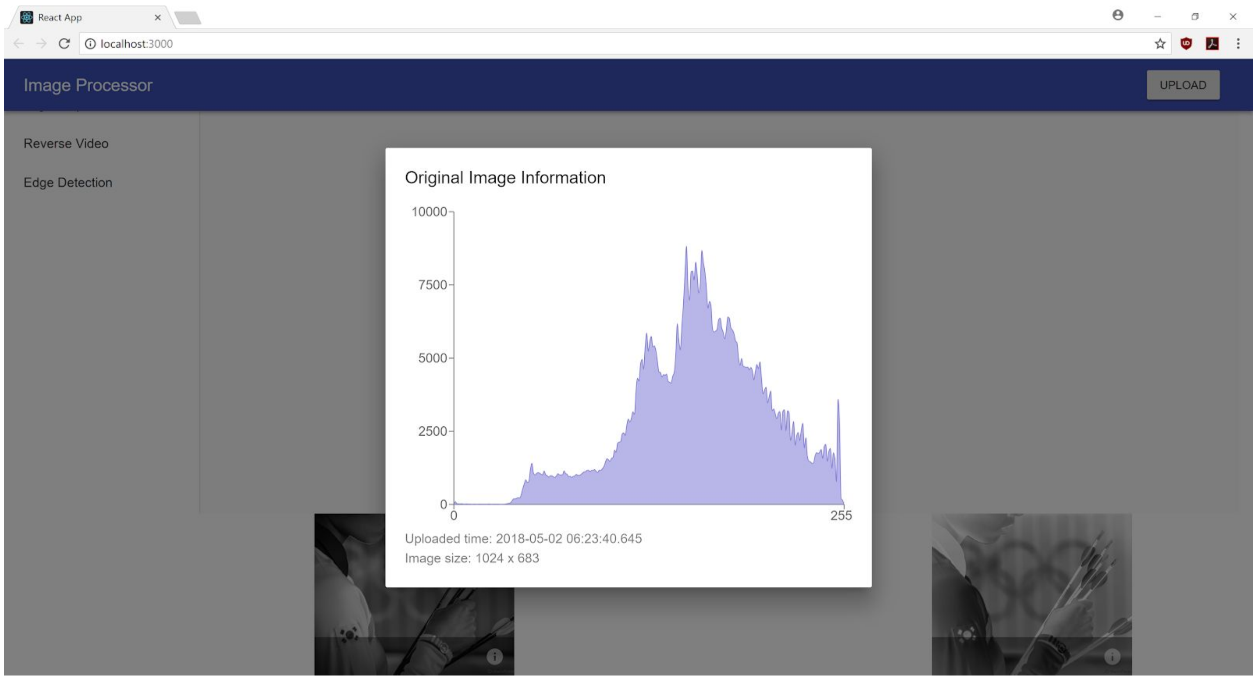
Name	Date	Type	Size	Tags	Date created
mult	4/30/2018 9:48 PM	Text Document	255 KB		4/30/2018 9:48 PM
Print trip and rec...	4/29/2018 2:07 PM	Adobe Acrobat D...	107 KB		4/29/2018 2:07 PM
ReturnTicketView	4/26/2018 5:24 PM	Adobe Acrobat D...	34 KB		4/26/2018 5:24 PM
737300-cropped	5/25/2013 12:22 AM	JPG File	1,905 KB		4/26/2018 10:44 AM
737300	5/25/2013 12:22 AM	JPG File	6,041 KB		4/26/2018 10:42 AM
Rio-0261-lesscro...	8/12/2016 4:31 PM	JPG File	1,387 KB		4/26/2018 10:29 AM
Rio-0261-cropped	8/12/2016 4:31 PM	JPG File	248 KB		4/26/2018 10:14 AM
gray_lc2	4/26/2018 12:23 AM	PNG File	6,250 KB		4/26/2018 12:23 AM
Picture1	4/25/2018 11:21 PM	TIF File	1,062 KB		4/25/2018 11:21 PM
FlamencoBeach_...	4/25/2018 11:17 PM	JPG File	389 KB		4/25/2018 11:17 PM
squishy-dog-che...	4/25/2018 10:58 PM	PNG File	506 KB		4/25/2018 10:58 PM
squishy-dog-che...	4/25/2018 10:57 PM	JPG File	78 KB		4/25/2018 10:57 PM
cloud-newsletter	12/10/2017 2:29 PM	JPG File	386 KB		4/25/2018 10:47 PM
a_nOfc-2yRNjBih...	4/25/2018 10:44 PM	JPG File	108 KB		4/25/2018 10:44 PM
c933bbd22e697...	4/25/2018 10:42 PM	JPG File	64 KB		4/25/2018 10:42 PM
localhost-152469...	4/25/2018 6:12 PM	Text Document	4 KB		4/25/2018 6:12 PM

File name:
All Files
Open
Cancel

Standard file selection dialog box opened by clicking Upload button



Original and processed image (reverse video)



Original Image Information

Uploaded time: 2018-05-02 06:23:40.645

Image size: 1024 x 683

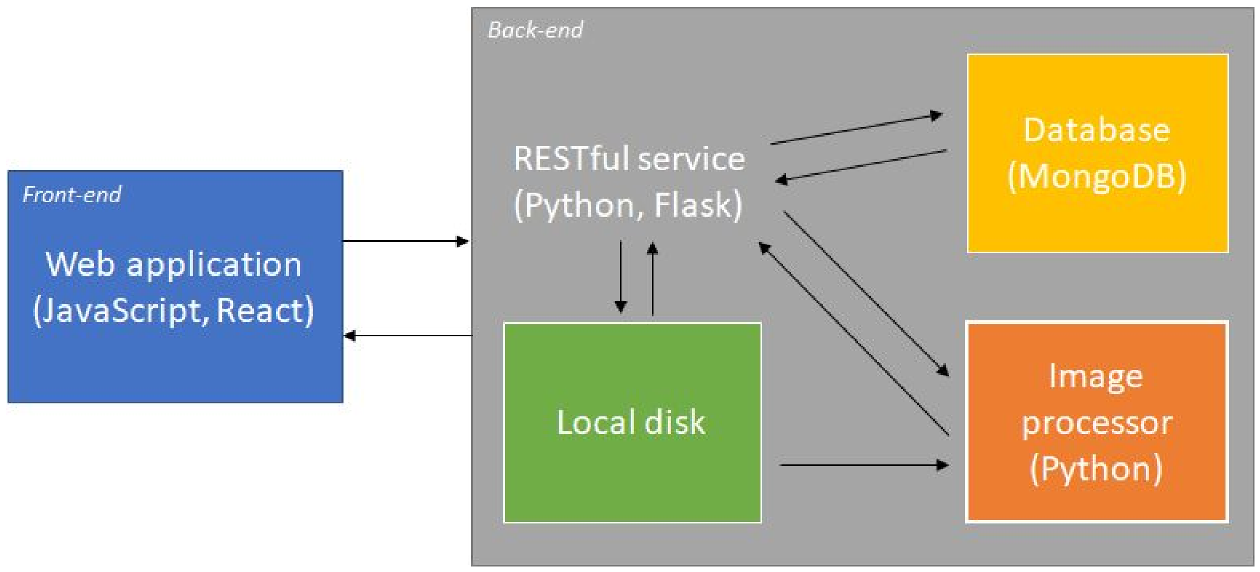
Information box of original image with relevant metadata

Architecture

Modify this section to best convey the technical design and architecture of what you're building. Some suggested subsections are below. Be sure to include what technologies are being used (and why if applicable).

Diagram

If it makes sense, draw a lucidchart diagram depicting how this service functions and what it depends on.



Endpoints

List endpoints and their interfaces here. Link to draft protobufs if possible for exact clarity on the API.

1. **Web application**

- JavaScript
- React (Library)

❖ Inputs
  ➢ **From client:**
    ■ User ID

        - Uploaded image files (tiff is supported for upload but not in browser rendering)
        - Processing command
    - **From back-end:**
        - String-formatted processed image files
        - Metadata
            - Histograms of input and processed images
            - Image dimensions
            - Base64 headers for jpg, tif, and png file types
            - Processing time for each image

- Outputs
    - **To back-end:**
        - String-formatted uploaded image files
        - Metadata
            - User ID/email
            - Command for image processing
            - Timestamp of upload
    - **To client:**
        - Uploaded image files
        - Processed image files
        - Metadata

The front end design language is based off Material UI and implemented using React. The user inputs their user email in the textfield on the side nav drawer which serves as a unique user identifier. The default processing command is histogram equalization. Other processing commands can be selected from the side nav drawer. Clicking the upload field brings up the system dialog for file selection. The upload is done using a Node JS library called ‘react-dropzone’. Images are converted to base64 and attached along with other relevant information and POSTed using axios.
The response to the POST request contains all relevant information needed for the display. Images are displayed in a grid list where each image forms its own tile. Each tile has an info icon that will bring up a dialog box with the relevant metadata (e.g. upload time, processing time, image size, histogram).

2. **RESTful service**

- Python
- Flask Framework (runs with gunicorn)
- Uses MongoModels for defining User class
    - Back-end server is only part of code to interact with database

- Inputs

        - **From front-end:**
            - String-formatted uploaded image files
            - Metadata
                - User ID/email
                - Command for image processing
                - Timestamp of upload
        - **From database:**
            - User object (MongoModels format)
- Outputs
    - **To database:**
        - User Objects (MongoModels format)
    - **To front-end:**
        - String-formatted processed image files
        - Metadata
            - Histograms of input and processed images
            - Image dimensions
            - Base64 headers for jpg, tif, and png file types
            - Processing time for each image

- Input from front-end is verified via type, key and value error checking. Raised errors caught by validation input are caught within the function. Function returns empty arrays along with a specific message detailing error if error occurs. Otherwise it returns validated parameters (email, command, etc.) for use later in code. In post function, if the email is empty, then a 400 error is returned with the specific message as a json.
    - Note: Timestamp should be sent as a string of format: "%Y-%m-%d %H:%M:%S.%f". This is converted to type datetime. User ID/email is string. Command is int between 1 and 5. Images are a list of strings.
- Try/except block to separate code for a new user and an existing user.
- Folder on vm is created called “images” which stores all user folders. Each user has a folder that holds both original and processed images.
    - Example:
        - Original input image = image0.jpg
        - Corresponding processed images = proc_image0.png, proc_image.jpg, proc_image0.tif (each input image has 3 paths to its processed images)
- Base64 strings of input images passed into post function are decoded and saved in the format indicated by the base64 header.
- Input command and timestamp are repeated to create list with length equal to number of input images. This ensures that each image saved to a user has an associated command and timestamp.
- Run image processing code. Extract data from returned dictionary (proc_data).
    - proc_data["processing_times"]

        - stat = proc_data["processing_status"]
        - orig_hist = proc_data["original_histograms"]
        - proc_hist = proc_data["processed_histograms"]
        - proc_data["processed_images"]
    - If all of the returned status codes are false, indicated no images were processed- a 400 error is returned with message.
    - All data is added to User instance in database (MongoDB)
        - Find user using email/id
        - Save paths of original images and processed images, command associated with each image, timestamp of each input image, status code indicated true = successful processing and false = processing failure
    - Encode all processed images in base64 to send back to front-end.
    - If status indicated failure, the base64 strings for all processed image types is empty. If this encoding fails, an empty array is returned and 400 error occurs.
    - Otherwise, json is returned to front-end with properties indicated above. 200 status code.
    - In future, code would be added to catch error caused if database is full.
    - No test function for function that saves numpy arrays. Functionality of the imsave() in scipy is well tested.

3. Image processor

- All image processing is implemented in Python using libraries such as scikit-image, NumPy etc.
- The image processor will receive the paths to the uploaded files and the processing command from the back-end.
- The files will be read from the local disk. Files that are not found or not able to be processed for any reason will not be processed and will have an associated processing status of False.
- The valid image processing commands and their associated functions are:
    1. Histogram equalization
    2. Contrast stretching
    3. Log compression
    4. Reverse video
    5. Canny edge detection
- Any invalid processing command will default to histogram equalization.
- Both color and grayscale images can be uploaded for processing.
- Any image larger than 1024 pixels (in either dimension) will be down-sampled while preserving the aspect ratio to enable efficient processing.
- Histogram data of the grayscale intensities will be created for both the uploaded and processed images.

- Upon completion of the processing, the following will be returned to the back-end:
  1. Message
  2. Filepaths
  3. Command
  4. List of processed images
  5. List of processing statuses
  6. List of processing times
  7. List of histogram data for uploaded images
  8. List of histogram data for processed images
  9. Image dimensions

Database

Be sure to clearly outline database schemas and future migration strategies if applicable.

We will be using a MongoDB database to store “User” MongoModels containing all paths to image files with their associated metadata. Each user will have a folder on the local filesystem of the virtual machine, where the uploaded and processed image files will be stored. “User” will containing the following information:

- ➢ User ID / Email
- ➢ Paths to uploaded images
- ➢ Paths to processed images
- ➢ Timestamps
- ➢ Processing command
- ➢ Processing status
- ➢ Processing time

Infrastructure

1. Where will instances of this feature/service run (within hospital networks, cloud, etc).

The back-end will be running in the cloud on our virtual machines. The front-end can be accessed by anyone connected to the internet.

2. Does this feature/service have specific infrastructure requirements?

This service does not have any specific infrastructure requirements beyond a device with a screen and internet connectivity.

Security Considerations

1. Does this service handle and/or persist PHI? If so, what precautions will be taken to safeguard this data?

Although this application can be applied to process images for many applications, it should be considered that users can use this application for processing patient images. Thus, we should try to store the patient data in the database via encryption (de-identification of the patient). We will consider MongoDB's approaches to encrypting data as stated on this webpage: https://docs.mongodb.com/manual/core/security-encryption/.

2. Does this service rely on secrets? How are those secrets distributed/safeguarded?

No, this application will not use information/source code that is private. The image processing routines that will be used do not need to be safeguarded. (In the case that encryption would be used to save patient data, the encryption code would need to be private. For the scope of this project, this will not be needed.)

Failure Modes & Mitigation

1. What are the service dependencies?

The service will depend on a RESTful web service which will be deployed on a virtual machine.

2. What problems occur if those downstream services are malfunctioning (high latencies) or down altogether?

If the services are malfunctioning, an error message will be displayed to the client.

3. What other failure modes exist for this feature/service, and what impact do those have? How can they be mitigated?

The database could run out of storage space, in which case an alert should be raised.

Alerting & Monitoring

1. What logging and monitoring will be in place for this feature/service

- Request times for different endpoints
- POST requests success/failure status
    - Logging is done at each point where an error is caught or not caught (success added via logging.info and errors recorded via logging.debug)

2. What alerts make sense for this service (if any) and how will those alerts be configured and set?

    - Server is not reachable, or slow and unresponsive
    - Database is full

    The alerts will be sent as email notifications to the maintenance team. Currently, these alerts have not been implemented and will be addressed in a future update.