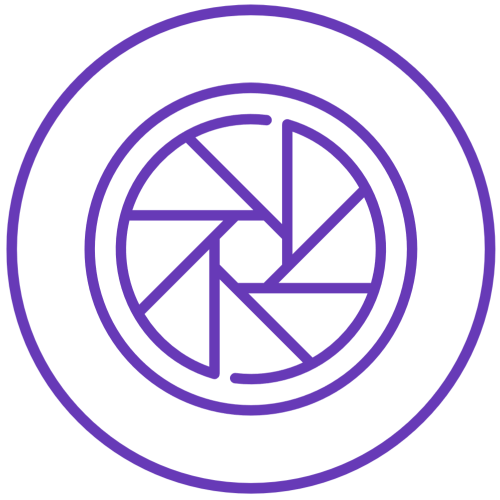
PicPro



CAB432 Assignment 2

Matthias Eder

11378093

Eric Zhang

10319191

23 October 2022

Contents

[Introduction 3](#_Toc118382154)

[Purpose & description 3](#_Toc118382155)

[Services used 3](#_Toc118382156)

[Sharp Node.js module (v0.31.1) 3](#_Toc118382157)

[Use cases 3](#_Toc118382158)

[US 1 3](#_Toc118382159)

[US 2 4](#_Toc118382160)

[US 3 4](#_Toc118382161)

[Technical breakdown 4](#_Toc118382162)

[Architecture 4](#_Toc118382163)

[Context Diagram 4](#_Toc118382164)

[Client / server demarcation of responsibilities 5](#_Toc118382165)

[Response filtering / data object correlation 7](#_Toc118382166)

[Features 11](#_Toc118382167)

[Image 11](#_Toc118382168)

[Transformations 12](#_Toc118382169)

[Presets 12](#_Toc118382170)

[Input/Output 13](#_Toc118382171)

[Scaling and Performance 13](#_Toc118382172)

[Test plan 15](#_Toc118382173)

[Difficulties and Reflexion 15](#_Toc118382174)

[Extensions (Optional) 16](#_Toc118382175)

[User guide 16](#_Toc118382176)

[References 16](#_Toc118382177)

[Appendices 17](#_Toc118382178)

[Appendix A 17](#_Toc118382179)

[Appendix 1A – Dockerfile Frontend 17](#_Toc118382180)

[Appendix 2A – Dockerfile Backend 17](#_Toc118382181)

[Appendix 3A – docker-compose.yml 18](#_Toc118382182)

[Appendix 4A – pm2 ecosystem.config.js 18](#_Toc118382183)

[Appendix 5A – Image “/” route 19](#_Toc118382184)

[Appendix 6A – Image “/fetch” route 20](#_Toc118382185)

[Appendix 7A – Image “/upload” route 21](#_Toc118382186)

[Appendix 8A – Image “/transform” route 22](#_Toc118382187)

[Appendix 9A – Preset “/” route 23](#_Toc118382188)

[Appendix 10A – Preset “/fetch” route 23](#_Toc118382189)

[Appendix 11A – Preset “/upload” route 24](#_Toc118382190)

[Appendix 12A – Transform function part 1 25](#_Toc118382191)

[Appendix 13A – Transform function part 2 26](#_Toc118382192)

[Appendix B 26](#_Toc118382193)

[Appendix 1B 26](#_Toc118382194)

[Appendix 2B 27](#_Toc118382195)

[Appendix 3B 28](#_Toc118382196)

[Appendix 4B 28](#_Toc118382197)

[Appendix 5B 28](#_Toc118382198)

[Appendix 6B 29](#_Toc118382199)

[Appendix 7B 29](#_Toc118382200)

[Appendix 8B 30](#_Toc118382201)

[Appendix 9B 30](#_Toc118382202)

[Appendix 10B 30](#_Toc118382203)

[Appendix 11B 30](#_Toc118382204)

[Appendix 12B 31](#_Toc118382205)

[Appendix 13B 31](#_Toc118382206)

[Appendix 14B 32](#_Toc118382207)

[Appendix 15B 32](#_Toc118382208)

[Appendix 16B 32](#_Toc118382209)

[Appendix 17B 33](#_Toc118382210)

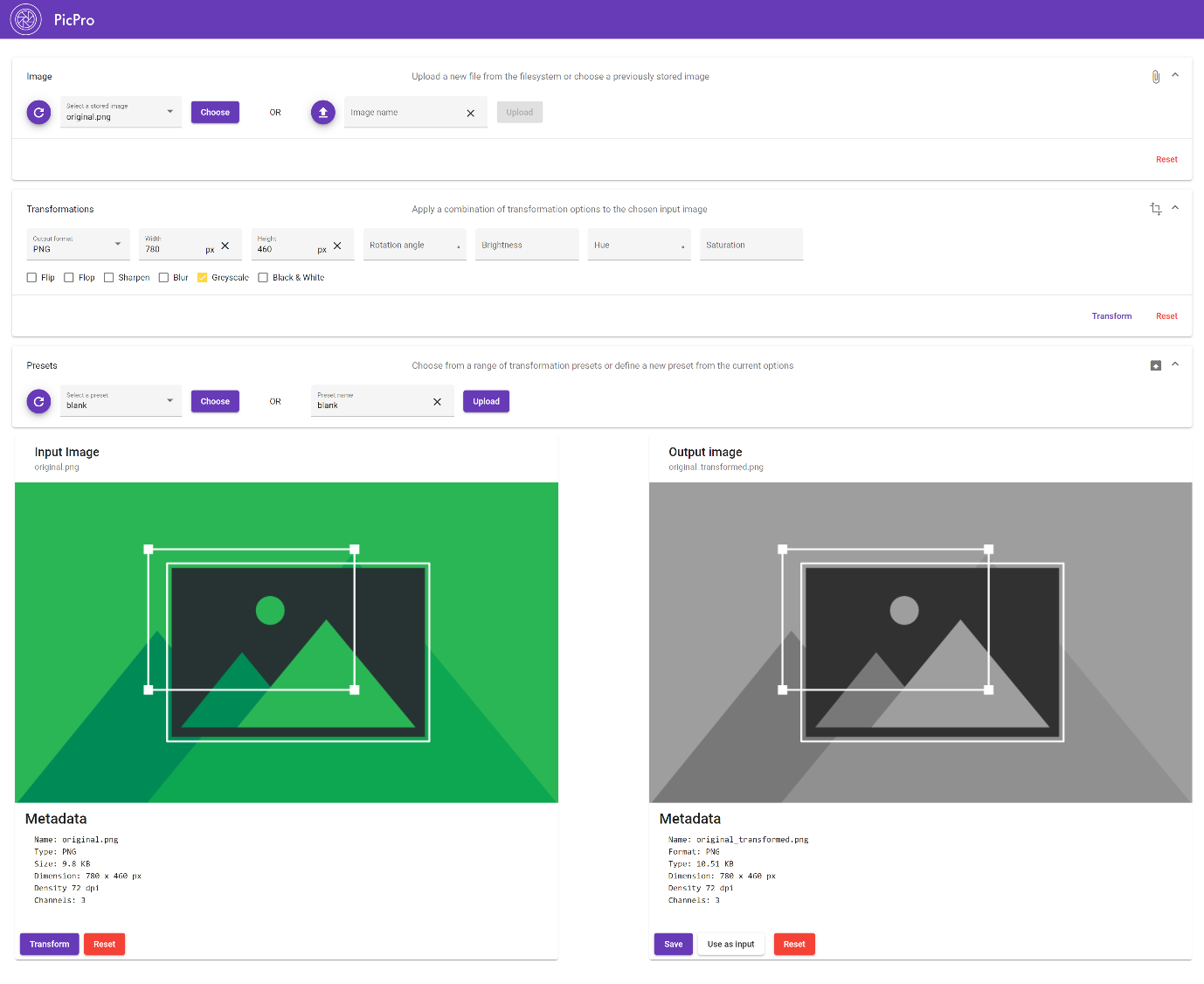
[Appendix 18B 33](#_Toc118382211)

[Appendix 19B 33](#_Toc118382212)

## Introduction

### Purpose & description

The PicPro Image Transformer allows users to alter images using various predefined options, such as changing the filetype format, resizing the image, or adding effects like blur and greyscale. It allows new and transformed images to be stored for users that wish to use the same image multiple times, along with saving presets for transformations so that users don’t need to repeatedly fill in the transformation values if they wish to use the same transformation over multiple images.



### Services used

#### Sharp Node.js module (v0.31.1)

The Sharp module accepts and reads an image file input to allow for various transformations to be done, such as filetype format, resizing the image, or adding effects like blur and greyscale etc.

Endpoint: N/A as it is not an API, but a Node.js module

Docs: <https://sharp.pixelplumbing.com/>

## Use cases

#### US 1

|  |  |
| --- | --- |
| As a | graphic designer |
| I want | to transform images like changing the format type or adding filters |
| So that | I can manipulate images to suit my design. |

#### US 2

|  |  |
| --- | --- |
| As a | graphic designer |
| I want | save previously uploaded images |
| So that | I don’t need to reupload the same image and can access it from the service with ease. |

#### US 3

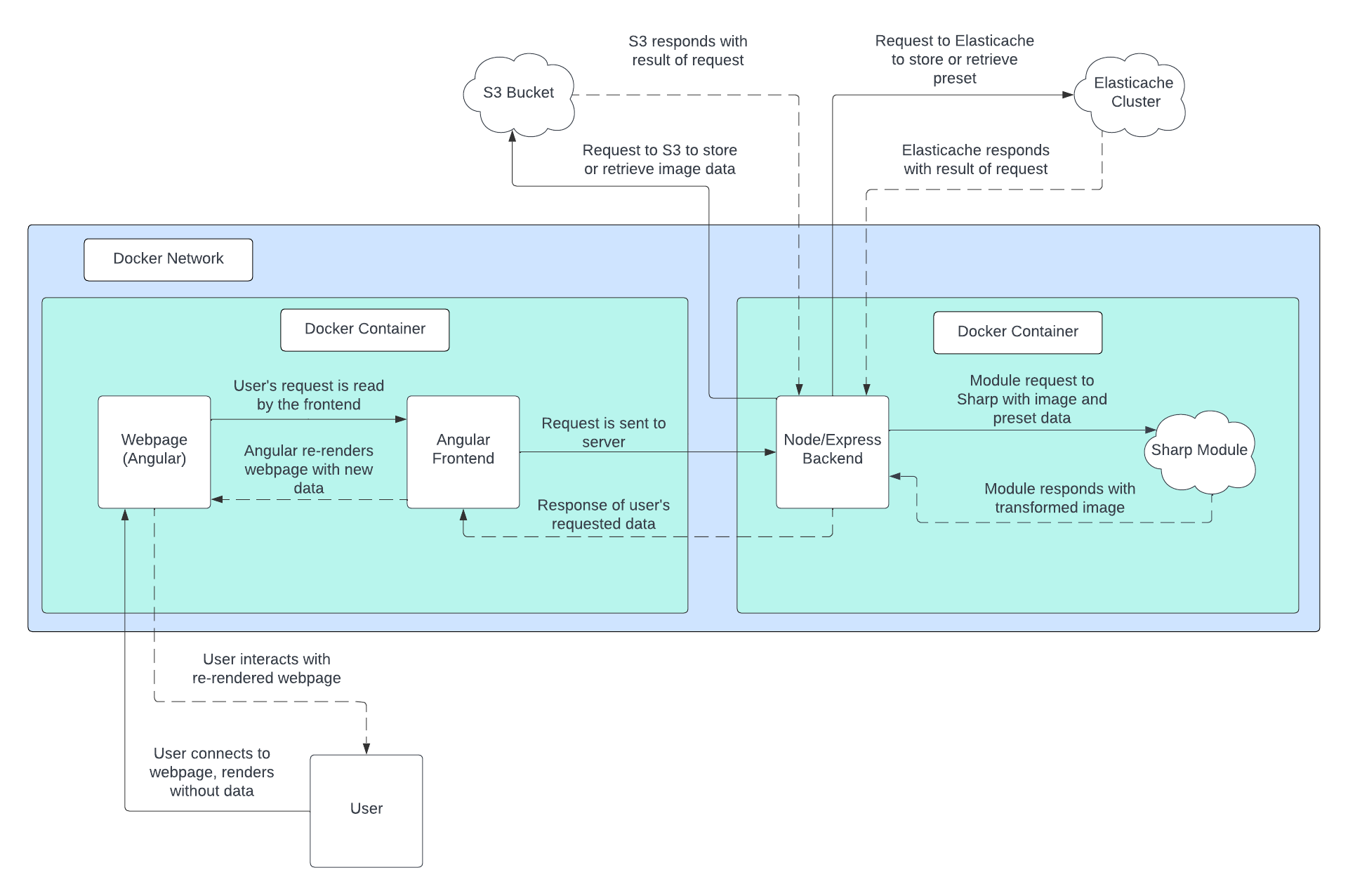
|  |  |
| --- | --- |
| As a | graphic designer |
| I want | save previous transformation presets/values |
| So that | I can apply the same transformation to multiple images without having to fill in the values every time |

## Technical breakdown

### Architecture

The flow of data can be seen in the context diagram below. The application consists of two docker containers, one containing the frontend, and one containing the backend. The frontend is a single page web application written with Angular, that processes and sends the user’s input data to the backend. The backend is an Express application based on Node.js, receiving the input data and filtering the desired attributes that will be used for either storing/fetching images via S3, storing/fetching presets via ElastiCache, or transforming images via Sharp. Once the response is received, the backend sends the output back to the frontend to be displayed to the user.

#### Context Diagram



These containers are built separately with *docker compose* with the Docker file content shown in linked Appendix. The [backend Docker file](#_Appendix_2A_–) sets the work directory to the backend folder, copies the package.json file from the node folder so that it can install the necessary dependencies using npm install. Afterwards, it copies the rest of the code in the container to run the complied *server.js* file to start the backend. This gets exposed to port 3000. The [frontend Docker file](#_Appendix_1A_–) follows similar steps to build the container, and then removes the default nginx website, which is the web service used for this application. The docker copies the dist folder from the build stage to the public nginx folder, while also copying the necessary ngnix config file that will be used. This gets exposed to port 80. These two Docker files are then used by the [docker compose file](#_Appendix_3A_–) to be built and tagged together, exposing them to the appropriate ports, 3000:3000 for the backend and 4200:80 for the frontend. Port 80 is needed so that the NGINX reverse proxy can listen on a separate port to redirect requests accordingly.

The way the application is deployed to AWS by pulling the images for frontend and backend from DockerHub, creating a Docker Network so that they can be accessed by container name and then running containers with the *--restart unless-stopped* flag set so that the containers are run on machine startup. This is needed for the dynamic creation of machines during scaling. The pm2 configuration file (ecosystem.config.js) shown in [Appendix 4](#_Appendix_4_–) runs the scripts for the 3 sections of the application: The docker network, the frontend container, and the backend container. This is used to run the docker application within the EC2 instance, allowing it to be started when the instance boots up for ease of scaling.

#### Client / server demarcation of responsibilities

The best way to describe the demarcation of responsibilities for the application is to again separate it into **frontend and backend**.

The **frontend** oversees the GUI (Graphical User Interface). What the users see on the webpage, from the header, to the three sections representing the Image, Transformation, and Preset features, along with the input/output images, are rendered from the frontend via Angular. The frontend also reads any input from the user so that it can be formatted inside a JSON request that gets sent back to the frontend, unpacking and re-rendering new output when the backend responds.

The **backend** handles the data processing. When the frontend makes a request, it sends it to the appropriate route, which can be categorised as the image and preset routes. The Appendix code has been linked below.

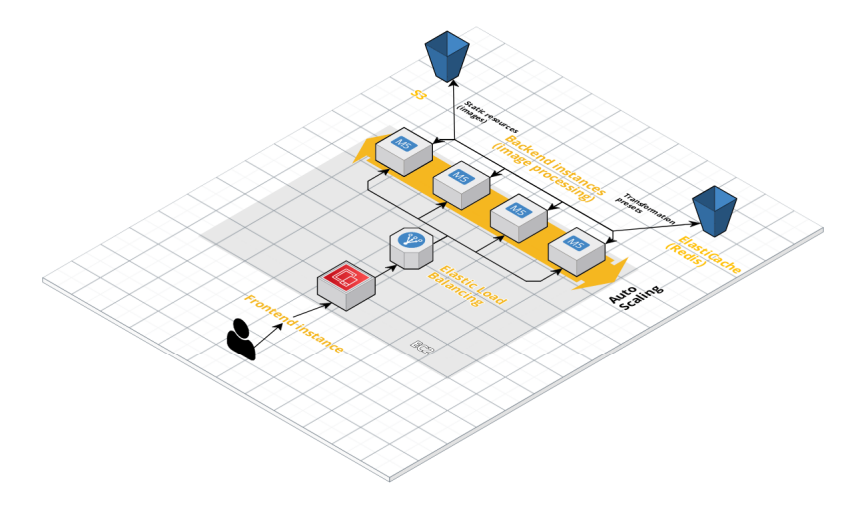
The **image routes** handle the requests from the Image and Transformation sections of the webpage, and there are 4 of them. The base route [“images/”](#_Appendix_5_–) retrieves the filenames of stored S3 images so users can choose one to fetch. The [“iamges/fetch”](#_Appendix_6_–) and [“images/upload”](#_Appendix_7_–) routes handle storing and retrieving images from the S3 bucket, fetch responding back the chosen image information that will be displayed on the webpage while upload only responds with the metadata. The [“iamges/transform”](#_Appendix_8_–) value again retrieves the desired image, using the transformation values sent from the frontend to apply the transformations via Sharp, before creating and sending the response containing the information for the newly edited image.

The **preset routes** handle the requests from the Preset section of the webpage, and there are 3 of them. These are similar to the image routes, and all of them interact with Redis via an ElastiCache Cluster: [“presets/”](#_Appendix_9_–) responds with the preset names that users can choose from, [“presets/fetch”](#_Appendix_10_–) retrieves the chosen preset that the frontend can use to fill in the Transformation section, and [“presets/upload”](#_Appendix_11_–) stores the current Transformation values along with a preset name provided by the user.

Looking at the architecture diagram below to explain the demarcation of responsibilities done via AWS, most of the application is contained within EC2. As can be seen, the user interacts with the frontend instance using the webpage, and this interacts with the backend via an Elastic Load Balancer. This scales the instances up and down depending on the computational load the application is receiving. The only difference between the proposal and the final architecture diagram is that the block representing the Elasticache cluster was replaced. Previously, the two persistence storages were represented with the same block, which was an S3 Storage block, but the final one replaces the right one with the Elasticache Database block.

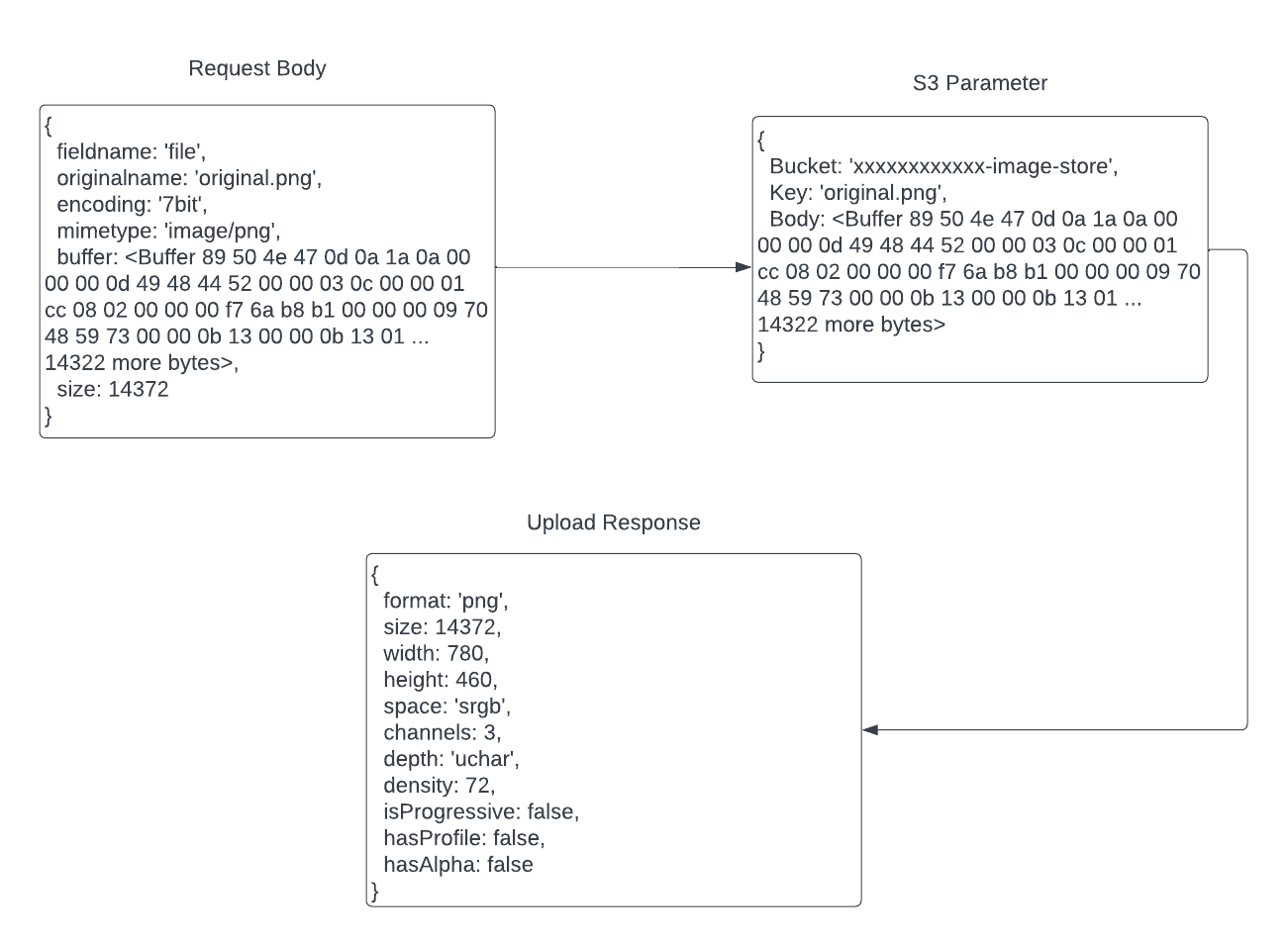
As shown in the diagram, the backend instances interact with the S3 bucket and the Elasticache Cluster when needed. S3 is a blob store, often referred to as object storage. It was chosen to store the images uploaded from the application due to how large and flexible it is, as users may upload several original and transformed images in quick succession. S3 can scale alongside the data stored in it, making it very suitable for large amounts of images.

ElastiCache for Redis is a memory-based cache and was chosen to store the presets that contain the transformation values due to its speed. As these presets are stored as JSON objects, they do not require the large flexibility in storage scaling that S3 boasts, and need to be accessed quickly, which ElastiCache provides very fast access time with very little latency, which is suitable for quickly retrieving the preset values that users will use to make their transformations.



#### Response filtering / data object correlation

The below diagram shows the data manipulation of the image [upload](#_Appendix_7A_–) route. The original request contains the image data, like the original name and the image buffer.



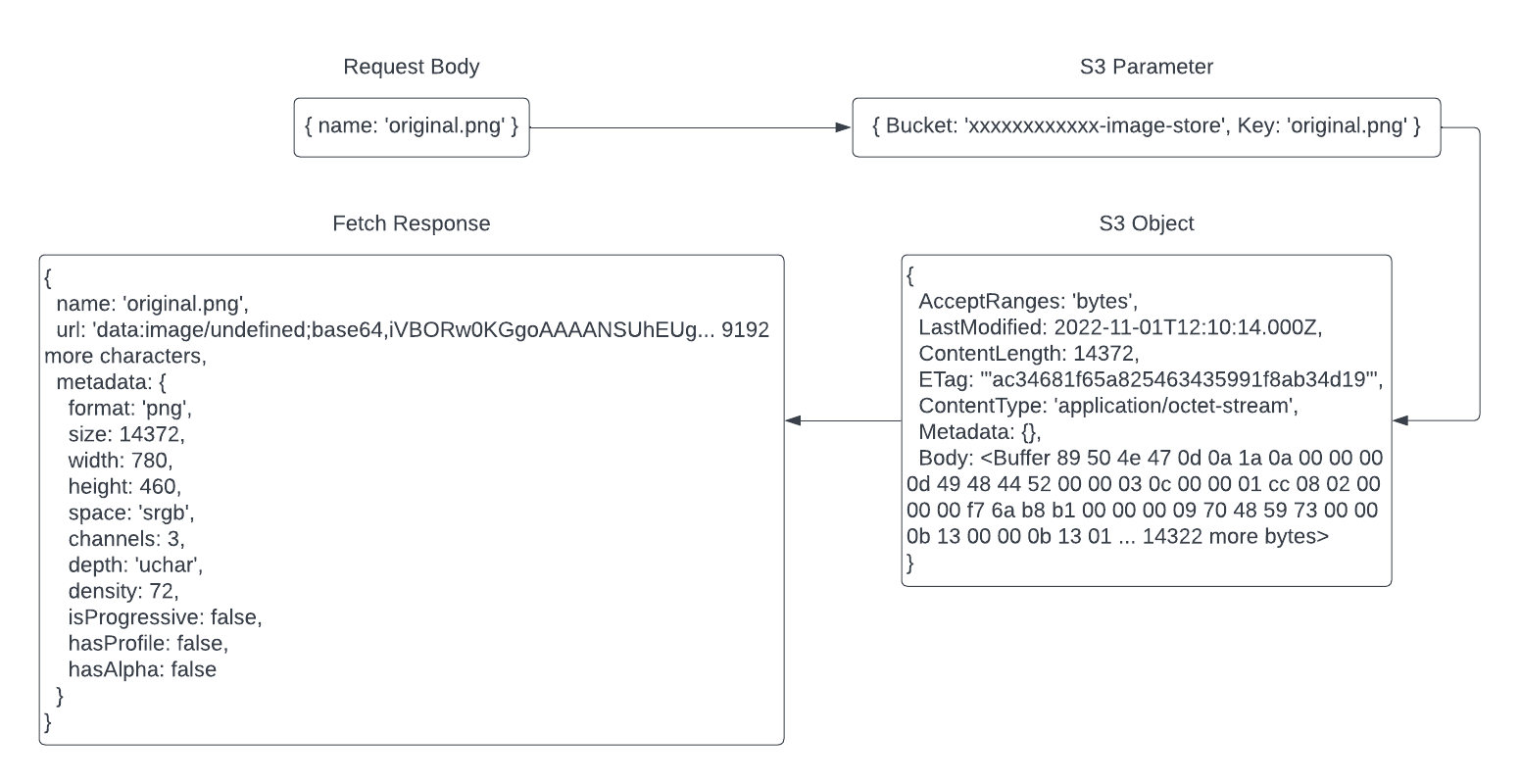
Buffer is used due to many file readers like the Sharp module often manipulating files as Buffer data, as they are suitable for fine-grained data manipulation, according to Digital Ocean **(2020, para. 3)**, along with its ease of conversion to other data types like a base64 URL, used to re-render the images to the webpage so they can be displayed back to the user.

The upload parameters contain the bucket name along with the filename from the request body and the image buffer from the image data. This gets uploaded to S3, while the response contains the metadata that is obtained using Sharp, as shown in the code segment below.

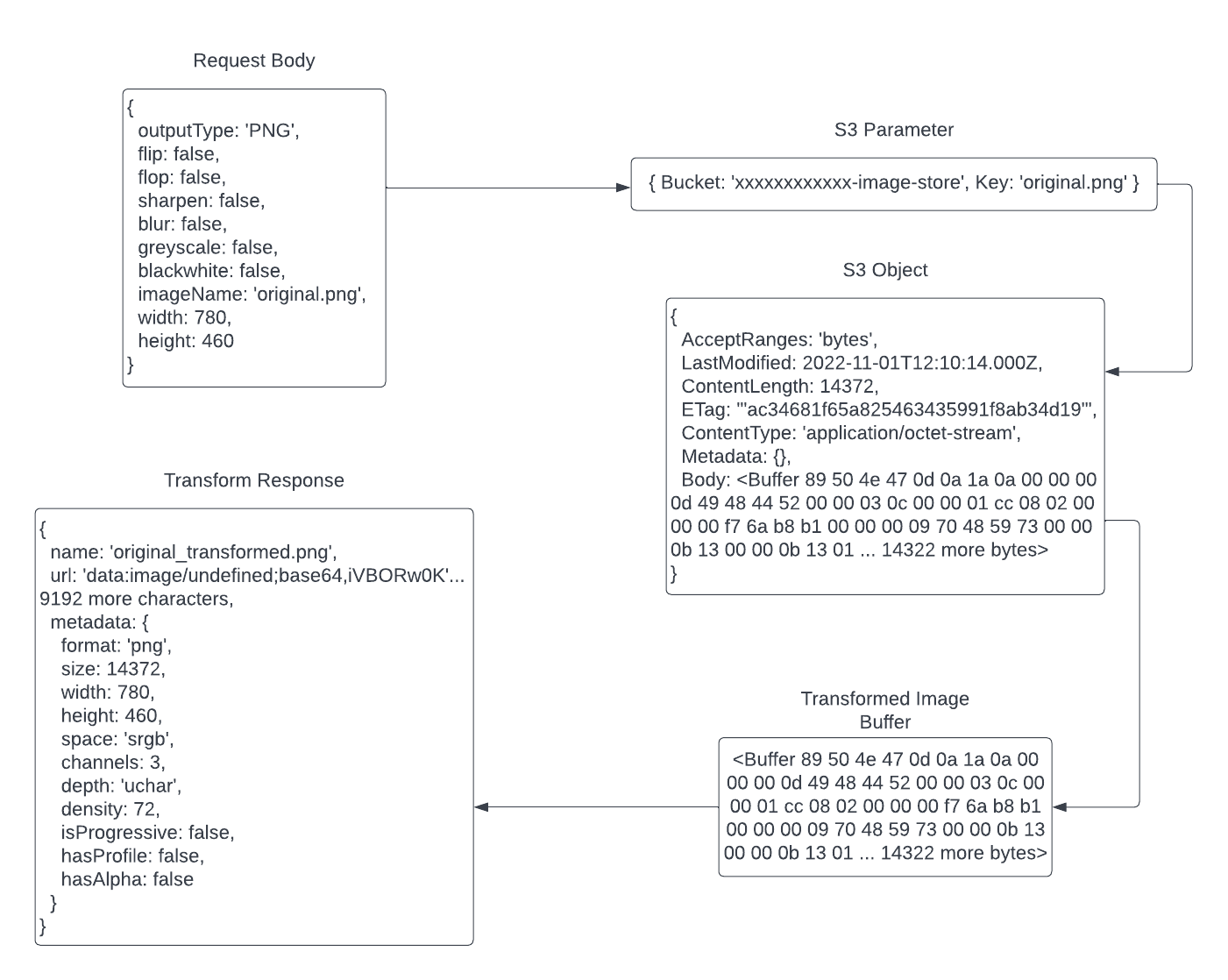


[Fetch](#_Appendix_6A_–) can be seen as going backwards from upload path. The original request is the filename of the image the user chose to retrieve. This filename is placed within a parameter JSON for S3, again containing the bucket name, which is used to retrieve the data. The object that is returned holds various information, like the last record of it being modified, and the body which contains the image buffer.

As metadata is not stored in S3 and is therefore blank, the route again uses Sharp to obtain it, which gets placed in the fetch response, also containing the filename and the image buffer which has been converted to a base64 URL.

[Transform](#_Appendix_8A_–) has a very similar path to fetch. Using the image name to obtain the desired image from S3, the request body containing the transformation values are sent to the transform function, that calls Sharp to apply the transformations using several if-statements to check through the request and apply the necessary transformations when values are supplied or marked as true.

The function first creates the Sharp pipeline by reading the image buffer and resizing based on the supplied width and height. If they haven’t been changed from the original image, then it just stays the same size. Every if statement afterwards stacks a prebuilt sharp function to the pipeline to apply the transformation, below is only a short example of the full function. The full example is shown in Appendix [12](#_Appendix_12A_–) and [13](#_Appendix_13A_–). At the end, it returns the pipeline containing all the transformations converted back into an image buffer.



The transform function returns the new image buffer for the transformed image. As seen in the code below, the original filename is split at the full stop so that ‘\_transformed’ can be added to differentiate the image from the original. This is only a default naming and when the user decides to download or save the image, they will be queried to supply a custom file name. The response is the same as previously shown in the fetch diagram, returning the new name, converted image URL and metadata obtained from Sharp.

When the input or output image is re-rendered back to the user, the response values are displayed in a HTML content card or mat-card like the example shown below. The filename is displayed in the subtitle html, below the card title “Input/Output Image”. The image is read from the URL in the img html, and the metadata is contained in the card contents, formatted with labels for each value.



Presets stay largely the same, a JSON object containing the transformation values, alongside a unique name or key. The values are displayed from HTML like what is shown below, [uploading](#_Appendix_11A_–) retrieving currently filled in values, and [fetching](#_Appendix_10A_–) filling in values from the preset. They are either shown with a mat form field like Saturation, most of them accept number type values, or a checkbox like Flip which simply displays if the transformation has been selected or not.



Examples of the preset JSON are explained in the [Presets](#_Presets) section of Features.

### Features

The flow of the data in our project, as well as its architecture, can be best described by the three main features of PicPro: Image, Transformations, and Presets

#### Image

Graphical user interface, application

Description automatically generated

The **Image** section allows users to either upload an image or choose from a set of stored images. Uploading an image creates a POST request to store them as a Buffer object in an S3 bucket. When accessing stored images, the keys are retrieved from S3 with a GET request so users can select which one to fetch (which uses the user-defined file names as keys, like “original.png”). Only fetching the file names to show in the selection element instead of the whole image data is intentional to reduce network load if the user decides to upload a new file instead. Thus, the actual data – which can be quite large depending on the picture – is only loaded once the user makes the intentional decision to load an image by clicking the “Choose” button. Once an image is chosen, a POST request is made to fetch the actual image data from S3, which is converted to a base64-encoded data URL and sent as a JSON response along with the filename key and metadata from the Sharp module. This gets re-rendered and displayed by the frontend to display the output to the user.

#### Transformations

Graphical user interface, application

Description automatically generated

The **Transformations** section allows users to select how they want to alter the chosen image. The values adjusted here are stored in a JSON request that gets sent alongside the data of your chosen image. Referring to the code example to the left, the transformation JSON contains the “imageName” which just stores a unique key for the requested image, and “transformations” that stores the values. The checkboxes are stored as Boolean values to identify whether they are to be applied to the image, and the text values get stored as string and number values as they control transformations that can be adjusted or have multiple options. Numerical values are optional so if the user does not define values there, they are also not sent to the server. Users can use the buttons on the bottom right to either apply the transformations or reset the values.

{

outputType: ‘PNG’,

flip: false,

flop: false,

sharpen: true,

blur: true,

greyscale: false,

blackwhite: false,

imageName: ‘original.png’

width: 5,

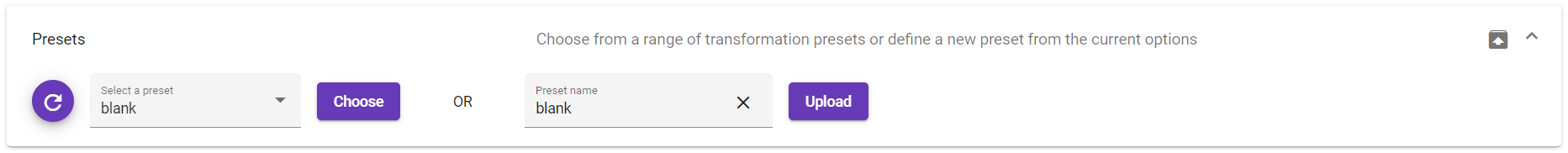
height: 5,

rotationAngle: 5,

brightness: 5

}

#### Presets



The **Presets** section allows users to store previously used transformations. Uploading a preset takes the values from the Transformation section, as can be seen in the code example to the left. The preset JSON is slightly different from the transformation JSON shown before, being separated into 2 parts: “name” which just stores a unique key for the transformation that can be used later, and “transformation” that stores the values. This is sent as a POST request to the backend that stores the preset JSON in an ElastiCache cluster using Redis. Any previously stored presets have their keys retrieved with a GET request, (like how filenames are displayed in the Image section) displaying the names of all the presets the user can choose from. The same reasoning can be applied here for the choice to only fetch names instead of the whole data as we wanted to focus on computational load while reducing network load. Once chosen, a POST fetch request is made to retrieve the desired preset, which gets sent to the frontend to replace the values in the Transformations section so that they can be used on an image.

{

name: 'blank',

transformation: {

outputType: 'JPEG',

flip: false,

flop: false,

sharpen: true,

blur: true,

greyscale: false,

blackwhite: false,

width: 5,

height: 5,

rotationAngle: 5,

brightness: 5

}

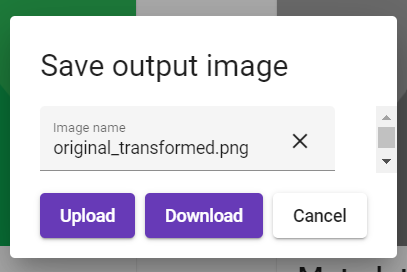
}

#### Input/Output

Graphical user interface, application

Description automatically generated

The images are displayed below these 3 sections, the original input shown on the left and the transformed output on the right. When a transformation is made, users have the option to save the new image, which brings up a popup dialog (shown below) that lets you choose a filename to either download to the local machine or upload to the S3 bucket. Users can also choose to use the output image again as the input image if further transformations need to be applied to the new image, or reset to use a new image or transformation.

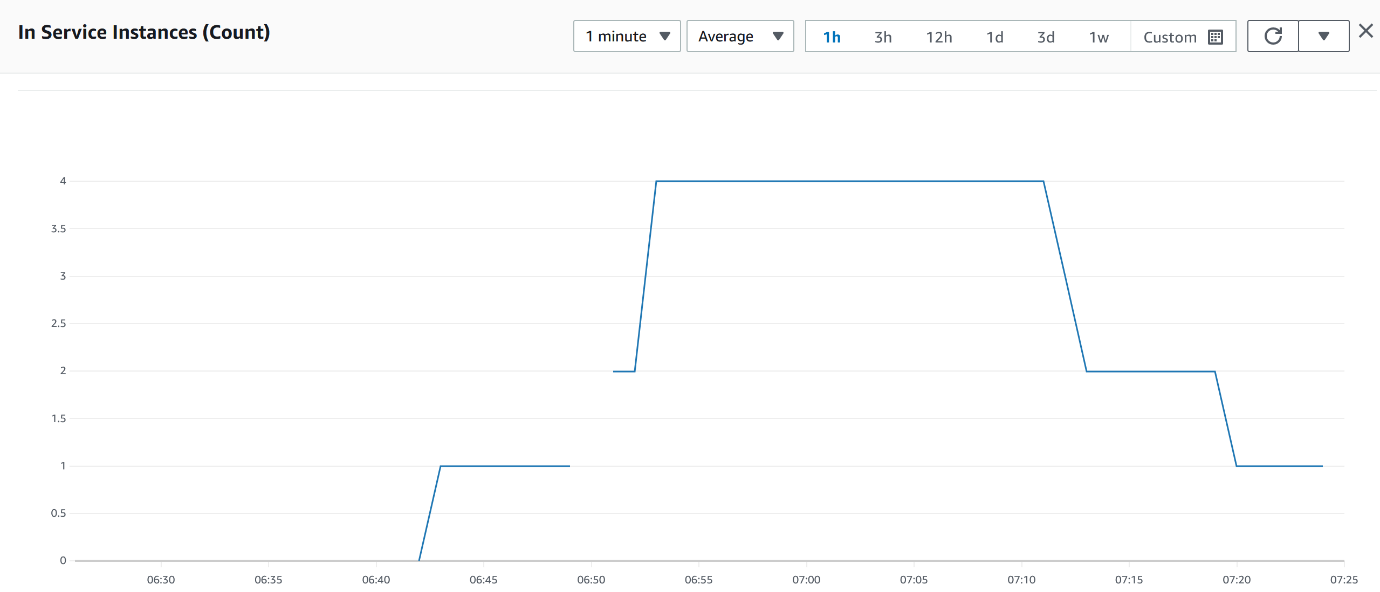


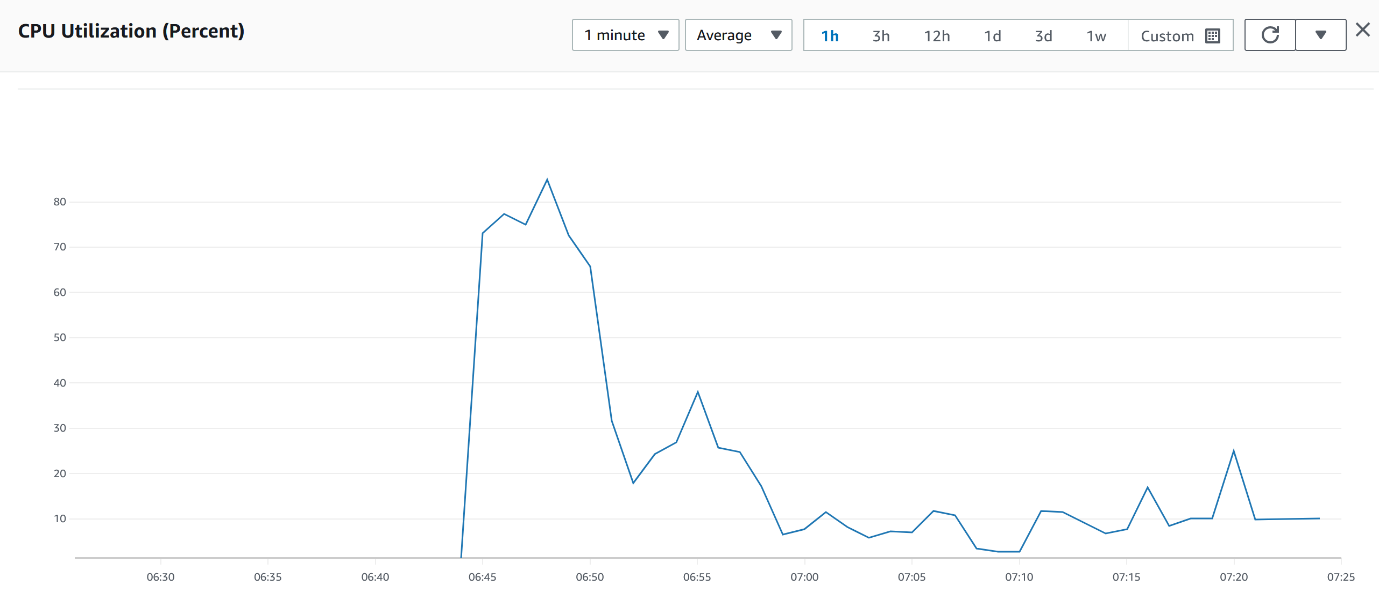
### Scaling and Performance

The application receives load from the computationally expensive image transformations and the amount of requests received. A few successive requests with relatively intense transformations like transcoding a large JPEG to a GIF and resizing to e.g. 8000px x 8000px will also max out the CPU capabilities of a single machine. While Sharp is very well optimised and usually performs a transformation pipeline in a few hundred miliseconds, using large input images with high resolution will easily create enough computational load from processing to allow easy scaling.

The scaling setup on AWS is very similar to the respective prac. Machines running frontend and backend are hosted on an LoadBalancer. The scaling policy in the AutoScalingGroup is set to use the avergae CPU utilization as the scaling metric and will scale out at a threshold value of 50%. We experimented with this value a while and found it easier to set it to a lower value for the sake of presentation and testing. A more flexible real-world setting with the potential of more resources would allow increasing the target value for CPU utlization, but we encountered some problems with higher values. If the ASG scaled out to the maximum number of machines quickly and further requests were sent, then the resources were also quickly maxed out resulting in Gateway problems. Investigating these issues were deemed to be out-of-scope for this assignment.

The screnshots shown below provide evidence of a proper deployment of our application in an elastic cloud setting with the ASG configured according to the above mentioned metrics. The first image shows the amount of in-use instances of the LoadBalancer over time. The second image shows the respective CPU utilization over the same time period. It can be observed that the CPU load increased rapidly at first before being cushioned by successful scaling-out. In this experiment, a Postman runner was setup to send 50 high-intensity transformation requests.





The cloud setup was configured according to our architecture diagram. A sidenote would be that in the diagram, the LoadBalancer is placed “behind” the frontend block while technically, a single machine will run both the frontend and backend containers. In case of scaling-out, the LoadBalancer will thus also create new instances of the frontend that could be accessed via their instance address and respective port. However, this was pictured intentionally to hide complexity as we will only interact with a single frontend instance and focus very much on data processing in the backend.

### Test plan

|  |  |  |  |
| --- | --- | --- | --- |
| Task | Expected Outcome | Result | Screenshot (Appendix B) |
| Image Select | Drop down displayed currently stored images | PASS | [1](#_Appendix_1B) |
| Image Upload | Loading bar shows during request, image file displays below | PASS | [2](#_Appendix_2B) |
| Image Fetch | Loading bar shows during request, stored image displays below | PASS | [3](#_Appendix_3B) |
| Image Section Reset | Input/Output images are empty again | PASS | [4](#_Appendix_4B) |
| Transform Values | User input is displayed in transformation fields/checkboxes | PASS | [5](#_Appendix_5B) |
| Transform Image | Transformed image appears as output below | PASS | [6](#_Appendix_6B) |
| Transform Reset | Transformation values are reset to the input’s original values | PASS | [7](#_Appendix_7B) |
| Preset Select | Drop down displayed currently stored presets | PASS | [8](#_Appendix_8B) |
| Preset Upload | Transformation values are stored, now available to select | PASS | [9](#_Appendix_9B) |
| Preset Fetch | Preset values fill in the above transformation values | PASS | [10](#_Appendix_10B) |
| Input Transform | Transformed image appears as output below | PASS | [6](#_Appendix_6B) |
| Input Reset | Input/Output images are empty again | PASS | [4](#_Appendix_4B) |
| Output Upload | Popup appears, image is saved with new name and is now available to select | PASS | [11](#_Appendix_11B), [12](#_Appendix_12B) |
| Output Download | Popup appears, image is downloaded with new name | PASS | [11](#_Appendix_11B), [13](#_Appendix_13B) |
| Output to Input | Input image is replaced with output image | PASS | [14](#_Appendix_14B) |
| Output Reset | Output image is empty again | PASS | [15](#_Appendix_15B) |
| No Image Input Disables Transform | Transform button is not highlighted and can’t be clicked/triggered | PASS | [16](#_Appendix_16B) |
| No Image Chosen Disables Fetch | Choose button is not highlighted and can’t be clicked/triggered | PASS | [17](#_Appendix_17B) |
| No File Chosen Disables Upload | Upload button is not highlighted and can’t be clicked/triggered | PASS | [18](#_Appendix_18B) |
| No Preset Chosen Disables Fetch | Choose button is not highlighted and can’t be clicked/triggered | PASS | [19](#_Appendix_19B) |

Please refer to the images provided in Appendix B to confirm the passing of the test cases. Some are hard to prove with a simple screenshot. We will be happy to show that these work in the demo.

Difficulties and Reflexion

The setup of the application was fairly straight-forward. We re-used most of the template Matthias implemented for the first assignment with regard to the frontend and backend projects, the dockerization especially for the NGINX webserver and reverse-proxying and the basic setup of the routing in the backend were re-used. The key points in implementation were to figure out in what way the image data can be stored efficiently in S3 so that few steps are necessary to upload and retrieve it. The connection and upload/download from and to S3 was also largely re-used. For Redis and ElastiCache it was mostly the connection and understanding how the cluster works that posed a slight challenge but was quickly resolved.

There aren’t any notable differences when comparing the proposal to the final product. The features and use cases mentioned in the proposal were successfully implemented.

When images or presets are uploaded, if the same name is used as a previously stored object, the stored object will be overridden by the new image or preset. There currently isn’t a warning in place to let users know they are about to overwrite a stored object, and this could be something to add in a future version.

As no external APIs were used, there weren’t any constraints there. The sharp module used to apply the transformations and retrieve metadata was relatively simple to learn and use. The application is however limited to what Sharp can do, so something like applying a mosaic blur would most likely require another module or API to add additional features for future versions.

When uploading presets, users can save them with a blank name, the current application does not require users to fill in the upload text field. This is something that should be fixed in a future version, detecting if there is text in the name field before allowing users to upload a preset

## Extensions

These are a few extensions that could possibly be added onto future versions of the application

* Allow users to have a private storage, uploading images/presets that can’t be accessed by others
* Add additional image transformation features, like mosaic blur, background removal, etc
* Allow the input and output image to be downloaded as a photoshop psd file, separating some of the transformations as separate layers for further editing
* Presets provide a preview of what transformations values them contain, so users don’t need to fetch them to see what they are
* Transformations provide a preview of what the image looks like after it’s transformed, so users can make more fine-tuned edits

## User guide

The “Features” section explains the components of our application in a visual way. In general, a user should either choose a previously uploaded image or upload a new one in the “Image” panel. Then the “Transformation” panel can be opened and settings can be made. The “Transform” button will trigger the processing and display the output image. The transformation settings can be saved in the “Presets” panel. Generally, the UI explains where the functionalities are contained and how it should be used.

For a guide on how to run the Docker containers, please refer to the README file(s) in the repository.

## References

Stack Abuse. (2020). Using Buffers in Node.js (Introduction). Digital Ocean. Retrieved October 28, 2022, from <https://www.digitalocean.com/commun1.ity/tutorials/using-buffers-in-node-js>

## Appendices

### Appendix A

#### Appendix 1A – Dockerfile Frontend



#### Appendix 2A – Dockerfile Backend



#### Appendix 3A – docker-compose.yml



#### Appendix 4A – pm2 ecosystem.config.js



#### Appendix 5A – Image “/” route



#### Appendix 6A – Image “/fetch” route



#### Appendix 7A – Image “/upload” route



#### Appendix 8A – Image “/transform” route



#### Appendix 9A – Preset “/” route



#### Appendix 10A – Preset “/fetch” route



#### Appendix 11A – Preset “/upload” route



#### Appendix 12A – Transform function part 1

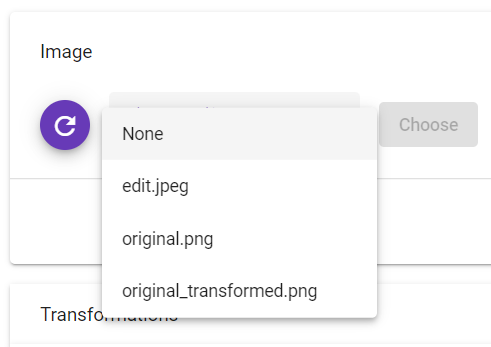


#### Appendix 13A – Transform function part 2



### Appendix B

#### Appendix 1B



#### Appendix 2B

Graphical user interface, application, website

Description automatically generated

#### Appendix 3B

Graphical user interface, application

Description automatically generated

#### Appendix 4B

Graphical user interface, text, application, email

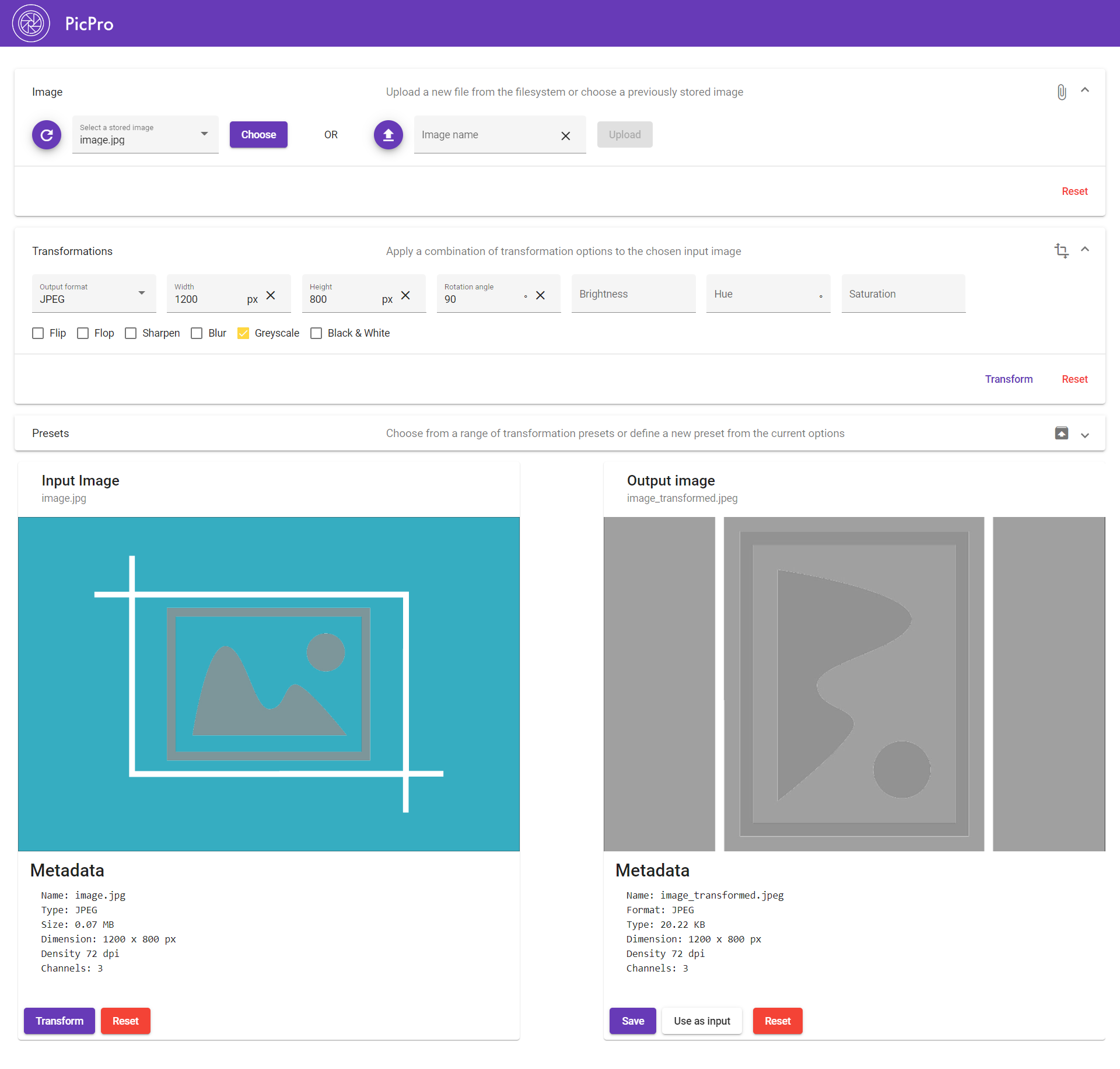
Description automatically generated

#### Appendix 5B

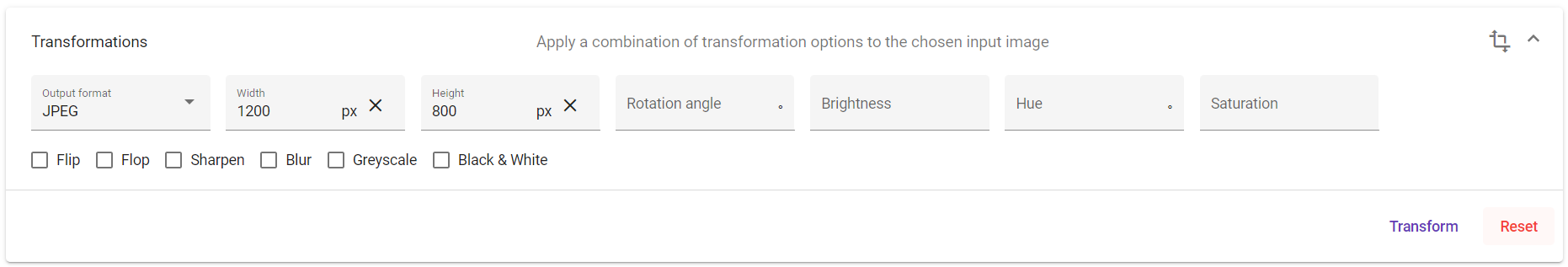
Graphical user interface, application

Description automatically generated

#### Appendix 6B



#### Appendix 7B



#### Appendix 8B

Graphical user interface, text, application

Description automatically generated

#### Appendix 9B

Graphical user interface, text, application, email

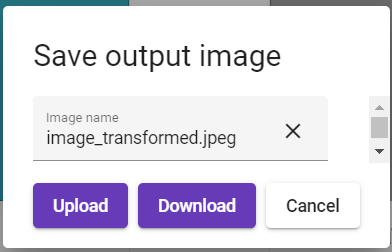
Description automatically generated

#### Appendix 10B

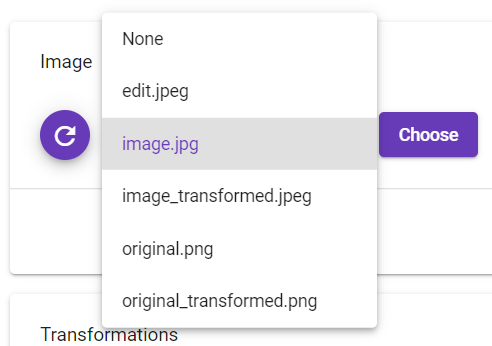
Graphical user interface, text, application, email

Description automatically generated

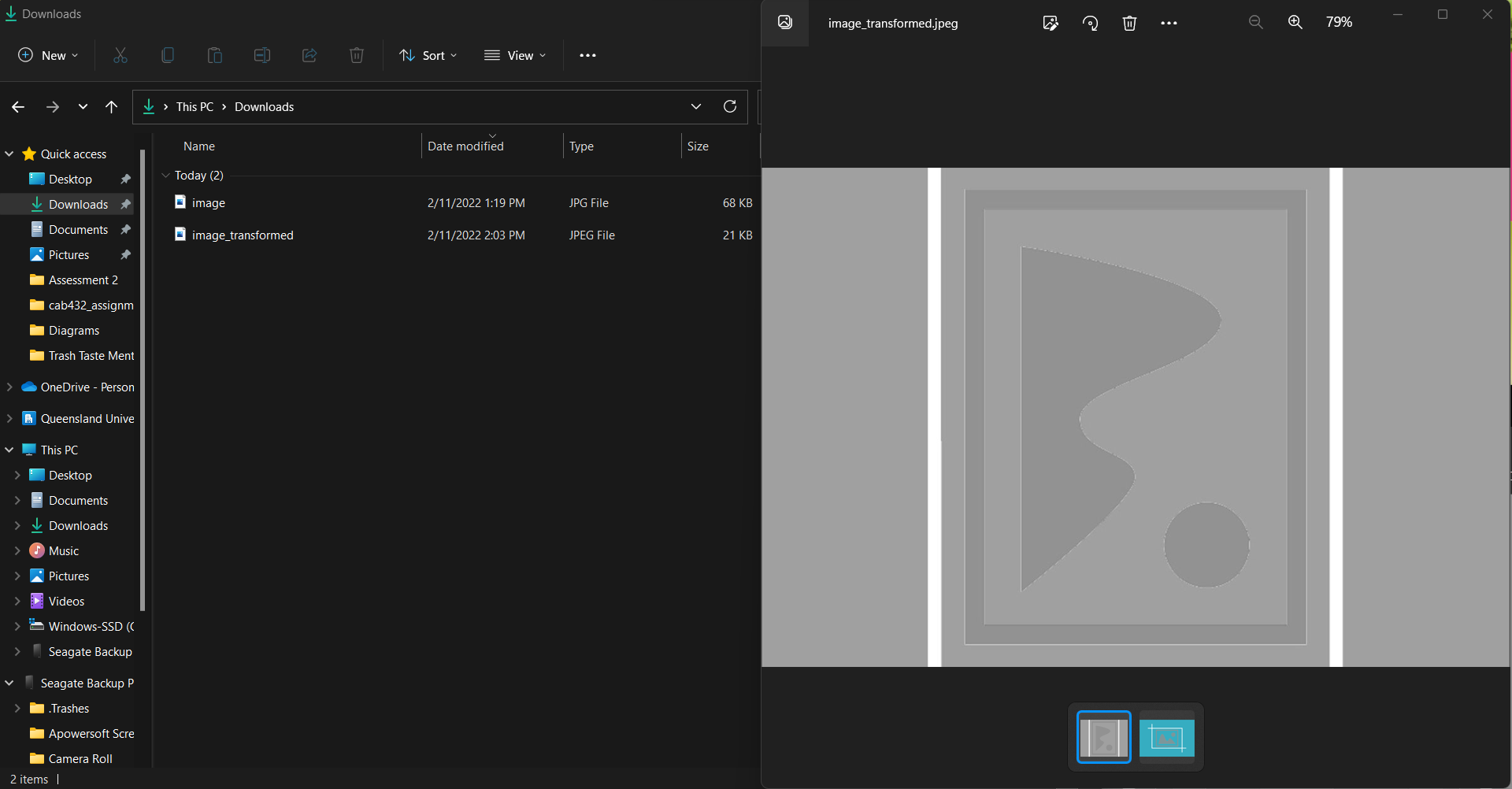
#### Appendix 11B



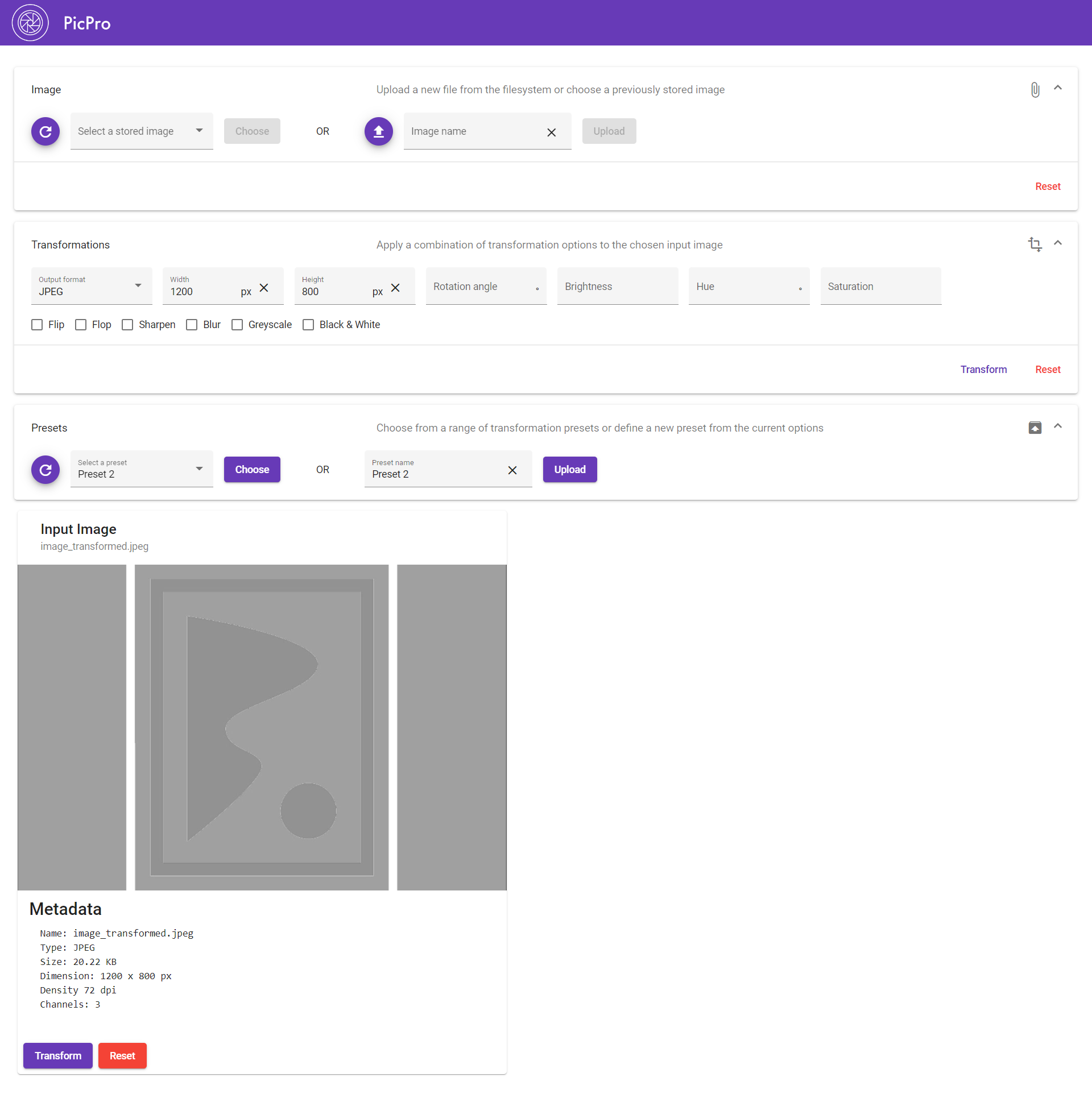
#### Appendix 12B



#### Appendix 13B



#### Appendix 14B



#### Appendix 15B

Graphical user interface, application, Teams

Description automatically generated

#### Appendix 16B

Graphical user interface, text, application, email

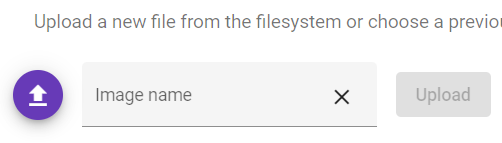
Description automatically generated

#### Appendix 17B

Graphical user interface, application

Description automatically generated

#### Appendix 18B



#### Appendix 19B

Application

Description automatically generated with medium confidence