Refiberd Tag Reader

Capstone Project Final Report, 2024

Team Members: Abdullah Azhar, Mustafa Hameed, Erin Jones, Isidora Rollan & Prashant Sharma
Date: May 3, 2024

Abstract

In collaboration with Refiberd, an early stage start-up in the climate technology space, our team built the Refiberd Tag Reader - a simple application to automate the capture of training data labels using machine learning. Refiberd has built an algorithm using computer vision in tandem with hyperspectral cameras to predict fabric compositions with >95% accuracy. To train their machine learning model, Refiberd must gather, take hyperspectral images of and capture the ground truth label for each fabric sample. The entire capture of their training dataset is currently done manually, taking precious time from their collaborators. Yet, one of their primary goals this calendar year is to increase the size of their training dataset from ~5,000 to ~20,000 images to further improve their accuracy. This is especially crucial for the algorithm's ability to identify minority compositions that don't occur regularly in their training dataset. Our solution aims to partially automate this process so it can be done by one person, reducing time spent recording sample labels. Though only a small part of the solution, our solution concomitantly supports Refiberd improving their accuracy, which is crucial to implementing their algorithm in the context of textile end-of-life. We succeeded in producing an MVP that will help Refiberd fight climate change - faster.

I. Introduction

Textile production is one of the most contaminating industries on the planet, contributing to 10% of global CO_2 emissions annually. According to the European Environmental Agency, 16-35% of microplastics released into the oceans come from synthetic textiles.[1] Moreover, less than 15% of used clothes and other textiles in the United States get reused or recycled. Once recycled, 80% of material is sorted, but less than 1% is recycled.[2] One of the challenges of recycling textiles is their composition. Most of our clothes are made from more than one fiber (cotton, polyester, linen, wool, etc). Moreover, according to Refiberd, most clothing and other garment tags are inaccurate. For example, a t-shirt tag may say "100% cotton", but it may have polyester threads sewing the different pieces together. This little detail can be crucial to deciding how to recycle it. Even in a simple garment, individuals often forget elements like stitching, logos and buttons that can render a garment ineligible for certain sustainable disposal mechanisms. Moreover, some garments can

[1] https://www.eea.europa.eu/publications/microplastics-from-textiles-towards-a

[2] https://www.nist.gov/news-events/news/2022/05/your-clothes-can-have-afterlife#:~:text=Only%20about%2015%25%20of%20used,climate%20change%20and%20pollutes%20waterways.

contain as many as 50 different fibers. When a manufacturer wants to reuse a textile, it must be sorted, purified, and reprocessed to create a new product. Sorting is one of the biggest challenges.

This huge gap in the industry motivated us to work on something that could help to solve this challenge, more specifically with computer vision techniques. The solution is non-trivial, especially when considering real world applications. Often fabric arriving at a recycling center may be damaged, wet, crumpled, etc. This renders most classification models developed from lab images or without a *massive* training dataset to be nearly useless. Additionally, as mentioned, the accuracy of such an algorithm has to be nearly perfect to effectively serve end-of-life facilities and recycling goals. As such, a hyperspectral camera that can capture more information that meets the eye is perfect for this task. Unfortunately, these cameras easily cost upwards of $50,000.

After exploring the option of building a hyperspectral camera or finding access to one to build a dataset and train a model from scratch, our team began to explore other options due to time and budget constraints. Many early-stage startups are already sorting waste and recycling using computer vision and a small subset are specifically focused on sorting textiles accurately. With the objective of contributing to this real-world problem, we decided to partner with one of these startups as a means of a head start, such that we weren't reinventing the wheel with inadequate time or resources. Initially, we were in contact with three companies: Sortile, based in New York City; Recycleye, based in London, UK; and Refiberd, based in Oakland, CA. All three use cutting-edge computer vision technology to identify and sort materials at end of life. Two of the three were focused on identifying the fabric composition of a garment.

After meeting with all three, we decided to continue with Refiberd. Introducing a partner into our capstone timeline could easily introduce entropy into our timeline and hamstring our ability to make an impact. The fact that Refiberd was located in Oakland helped to alleviate concerns related to partnering with an outside organization. The Refiberd Tag Reader came as a solution to streamline the process of labeling new training data so that Refiberd can continue to increase the accuracy of their machine learning model with less effort. We envisioned an end-to-end application that captures a photo using a mobile device, extracting necessary composition information from a textile sample tag in an automated fashion. Once the user confirms that the tag was read correctly, this information is saved in a database, overall reducing the time and effort needed to capture ground-truth for their dataset. The manual effort associated with this task is discussed in detail below. The front-end of the application is simple, as most of its process happens in the backend, where a model processes the image and interprets the results, and the backend saves this information in a database. Additionally, per client requirements, this app needed to be built using Amazon Web Services (AWS), to integrate appropriately with their organization's infrastructure.

II. Refiberd

Refiberd is a small start-up based in Oakland.[3] It was founded by Sarika Bajaj and Tushita Gupta, who were eager to solve the global textile waste crisis. The problem of textile recycling has many steps, one of which is identification of a fabric's composition in the context of sorting. This can help recycling plants sort and recycle correctly and help other companies identify garment composition. For this matter, Refiberd uses cutting-edge technology that leverages a hyperspectral camera to take a picture of a textile and a machine learning model that retrieves the exact composition of it. Hyperspectral imaging is an analytical technique based on spectroscopy that collects many images at different wavelengths for the same spatial area. Where the human eye sees three color channels, blue, green, and red, hyperspectral cameras aim to measure a near continuous spectrum of light and can have hundreds of channels. Each channel offers new information and thus improves model accuracy and generalizability.

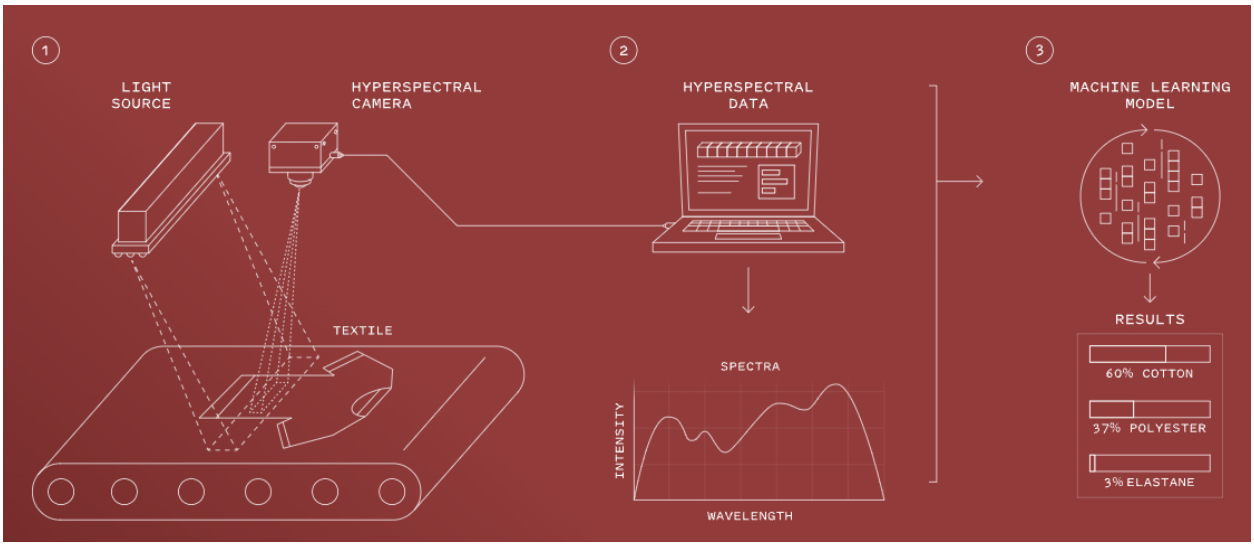

Figure 1 - Process to identify fabric composition.[4]

As can be seen in Figure 1, the process is that the hyperspectral camera (1) captures lines of hyperspectral data from the moving textile at a predetermined frame rate and (2) the computer assembles these lines to form a hyperspectral cube. Finally (3) a machine learning model processes this cube and predicts the material composition.[4]

III. Our Problem

Building a machine learning model has its challenges. One of them is to have quality data to train the model and be able to fine-tune accordingly. In a universe where many natural and artificial fibers exist and infinite combinations of them are present in samples, training the Refiberd ML

[3] https://refiberd.com/
[4] https://refiberd.com/technology/

model is a big challenge. In order to do this, they search for reliable providers of fabrics from which they can trust that the fabric composition declared in the tag is true.

Process

Once they have these samples in their Oakland office, they follow these steps to log and capture the fabric sample (Table 1):

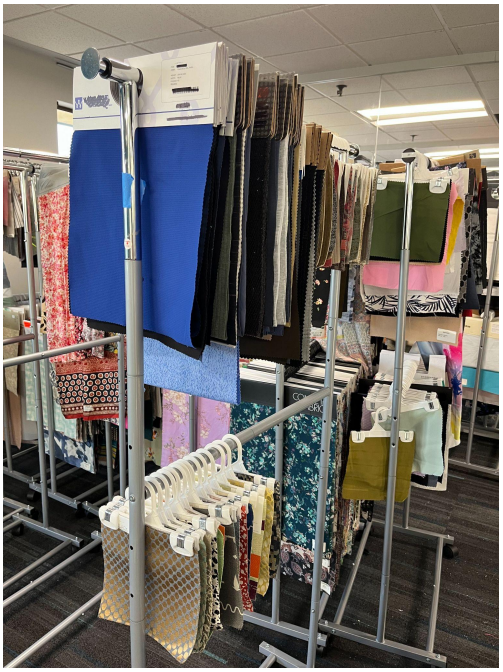

Figure 2 - Samples placed on racks at Refiberd's Oakland office.

1. Hand sort samples, quickly removing those with compositions not currently supported by their algorithm
2. Hang them in a rack (figure 2)
3. Add a sticker with a sample ID, which is currently handwritten
4. Add the composition distribution to a Google spreadsheet.

In the fourth step, two of four Refiberd employees are needed to log the label data. One reads the tags placed in the rack and says out loud the composition of the sample while the other writes it down in the Google spreadsheet. **On average, this process from steps 2 through 4 are estimated to take two workers 1.5 - 2 minutes.**

With a team of four people, half must spend entire days to label all the samples they receive. Then, a third team member takes pictures of the labeled samples with the hyperspectral camera. When we visited them in February, Refiberd was aiming to to label and photograph 15,000 new samples (300% increase in training set size) to improve accuracy in their identification of fabrics.

Person/Steps	Step 1 - Sorting	Step 2 - ID	Step 3 - Labeling	Step 4 - Auditing
Person 1	Sort samples (from boxes)	Paste sticker with a handwritten unique ID on tag	Read out loud fabric composition from the tag	
Person 2	Hang them in racks	Write ID in a new row on Google Sheet	Write down person 1's dictation in a spreadsheet	
Person 3				Review the tag and the recorded results

Table 1 - Current process for labeling new samples.

Tags

The pictures shown in Figure 3 evidence the lack of standardized structure or method for naming and abbreviating fibers within the manufacturer's textile tags. This adds an extra layer of complexity, even for the human eye. For example, an abbreviation of "T" is usually known as Tencel, but when the fabric comes from China, "T" is Polyester. Our algorithm would not only need to identify composition information from a slew of other information included across the tags, but it would need to identify and translate certain categories to match the classes needed by Refiberd (e.g. Tencel to Polyester).

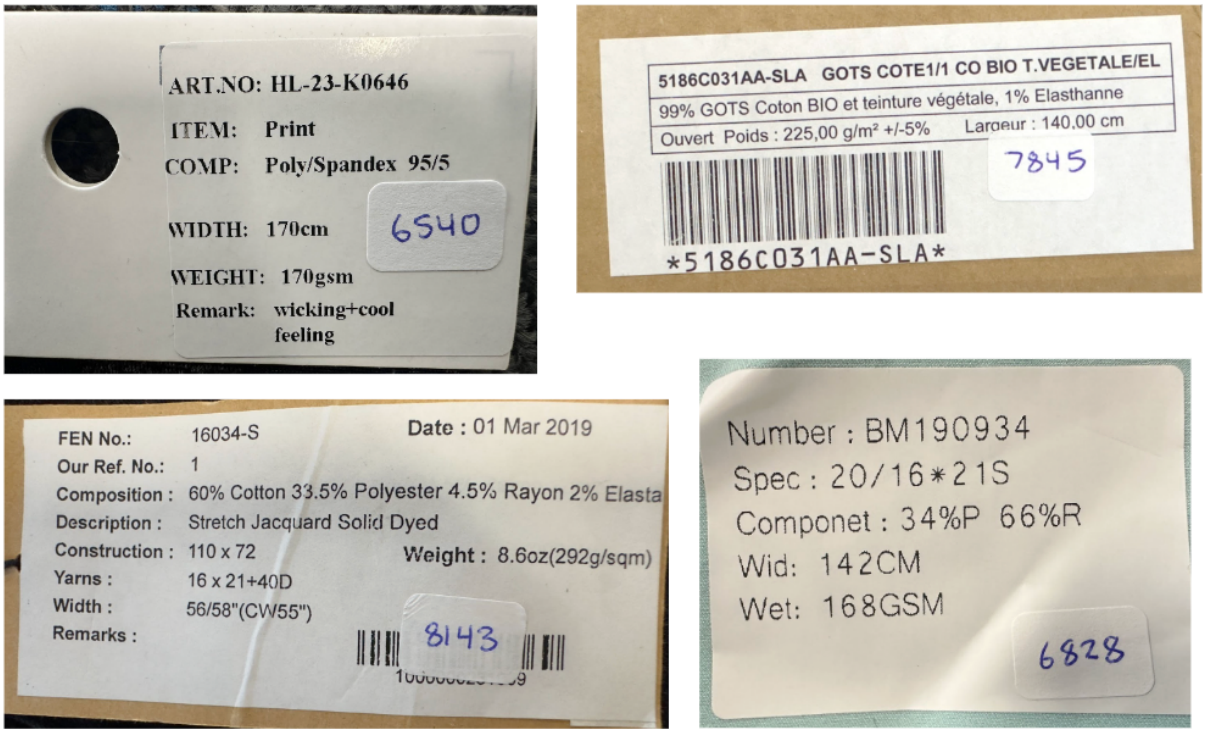

Figure 3 – Samples of tags; Note lack of standardization.

Refiberd Requirements

Refiberd asked us to find a way to automate the process of labeling new data, as we hoped to work on solving a problem for them that leveraged computer vision. In consultation with Refiberd, our team identified the following requirements:

1. Capturing and labeling only requires one person.
2. Should cloud technology be used, it must be hosted in AWS.
3. The major goal relates to ensuring the process of capturing the label via the application is cheaper than having two individuals completing the labeling.
    a. Note: If the process time remains the same, but the labor requirements are reduced to a single person, the solution is still recognized as cheaper due to the saved labor cost associated with freeing up another person.

4. As implied by requirement 3, the end-to-end process should take at most 2 minutes per tag.

We committed to delivering a minimum viable product to Refiberd by the end of our capstone project.

IV. User Experience

In crafting an end-to-end user experience that prioritizes simplicity and speed, we closely examined the requirements and the existing workflow for labeling new data. Our goal was to ensure that users can achieve their objectives with minimal distractions: capturing an image of the tag in landscape mode, reviewing the model's predictions, and saving their input.

Optimal Workflow (Happy Path): The user logs in, captures an image, and confirms the fabric composition. This process is repeated smoothly until all samples in the batch are successfully processed.

Less Ideal Workflow (Not So Happy Path): The user logs in and takes a photo. If the system encounters an error, the user may need to retake the photo and reconfirm the fabric composition before proceeding.

Challenging Workflow (Unhappy Path): After logging in and taking a photo, the user must manually edit the information and confirm the changes if the model's prediction is incorrect.

Our approach involved exploring various pathways to streamline the user’s interactions, ensuring they can complete their tasks efficiently and effectively, regardless of the path they encounter. We aimed to make the experience as intuitive as possible, reducing the need for extensive troubleshooting or intervention, thus accommodating all potential user scenarios.

Figure 4 represents the end-to-end user experience design. This was created alongside several wireframes in Figma, which were all presented to Refiberd prior to diving into the application development.

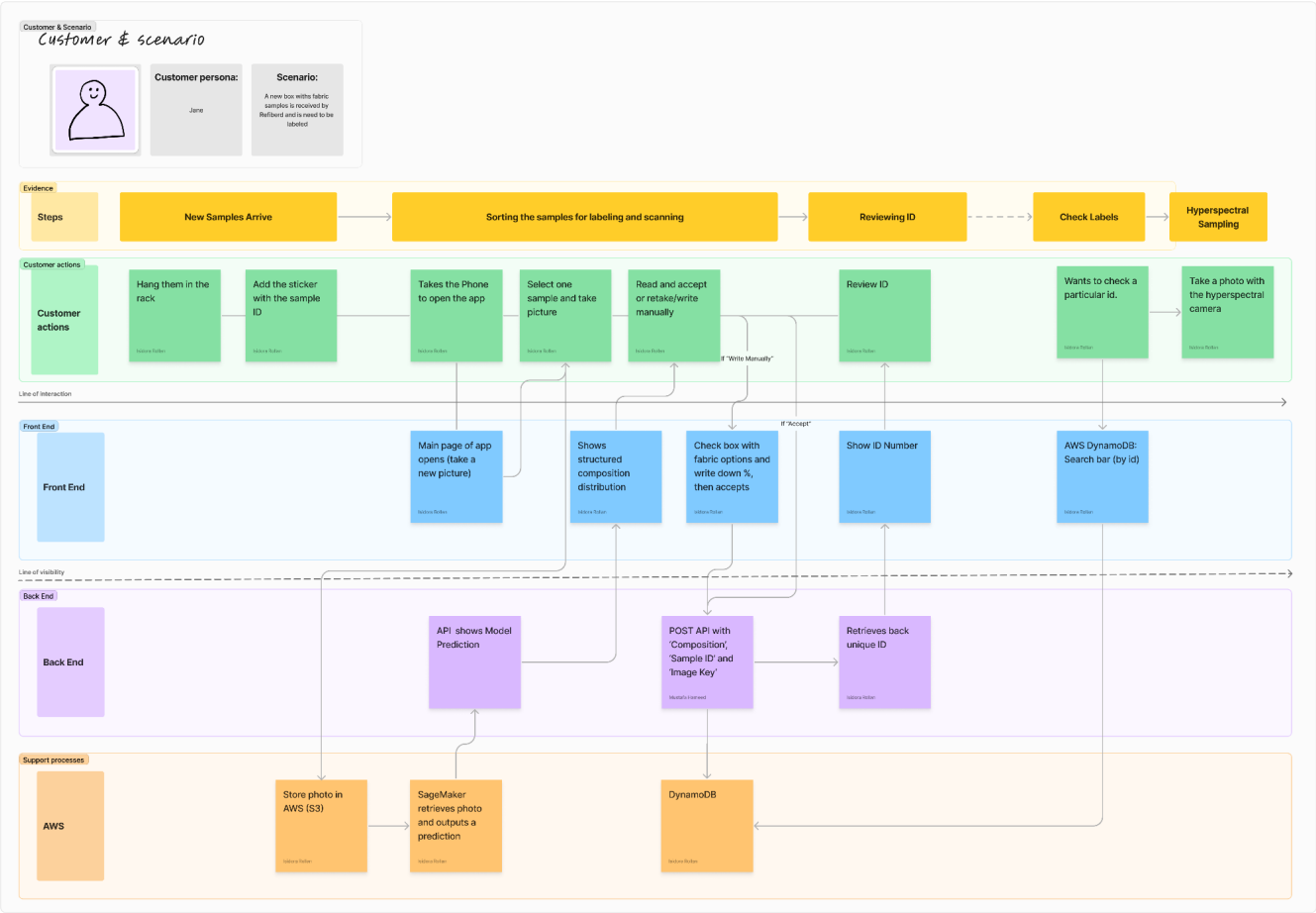

Figure 4 - End-to-End User Experience Design.

V. The Solution

Architecture Overview + Introduction

Much of the application, apart from the fine-tuned model itself, is hosted via Amazon Web Services (AWS), which was already in use by Refiberd. AWS hosts a suite of services that enabled the end-to-end creation of an application, powered by a machine learning model, without the need for any local hosting or deployment. This showcases the power of AWS as a service.

Our team saw the opportunity to learn and deploy using AWS as one of the major learning opportunities encompassed by this project. Due to cost, cloud services such as AWS are not often taught in courses, but they are used widely across industry, presenting us with the opportunity to gain valuable insights on this platform.

The finished architecture successfully hosts an application, which allows the user to take a picture of a fabric sample tag on their phone, send the picture to the model for processing, verify the composition produced by model output and save the results in a cloud database. The AWS

architecture, shown in the figure below, leverages a total of six AWS services and the Hugging Face hub (which hosts the machine learning model).

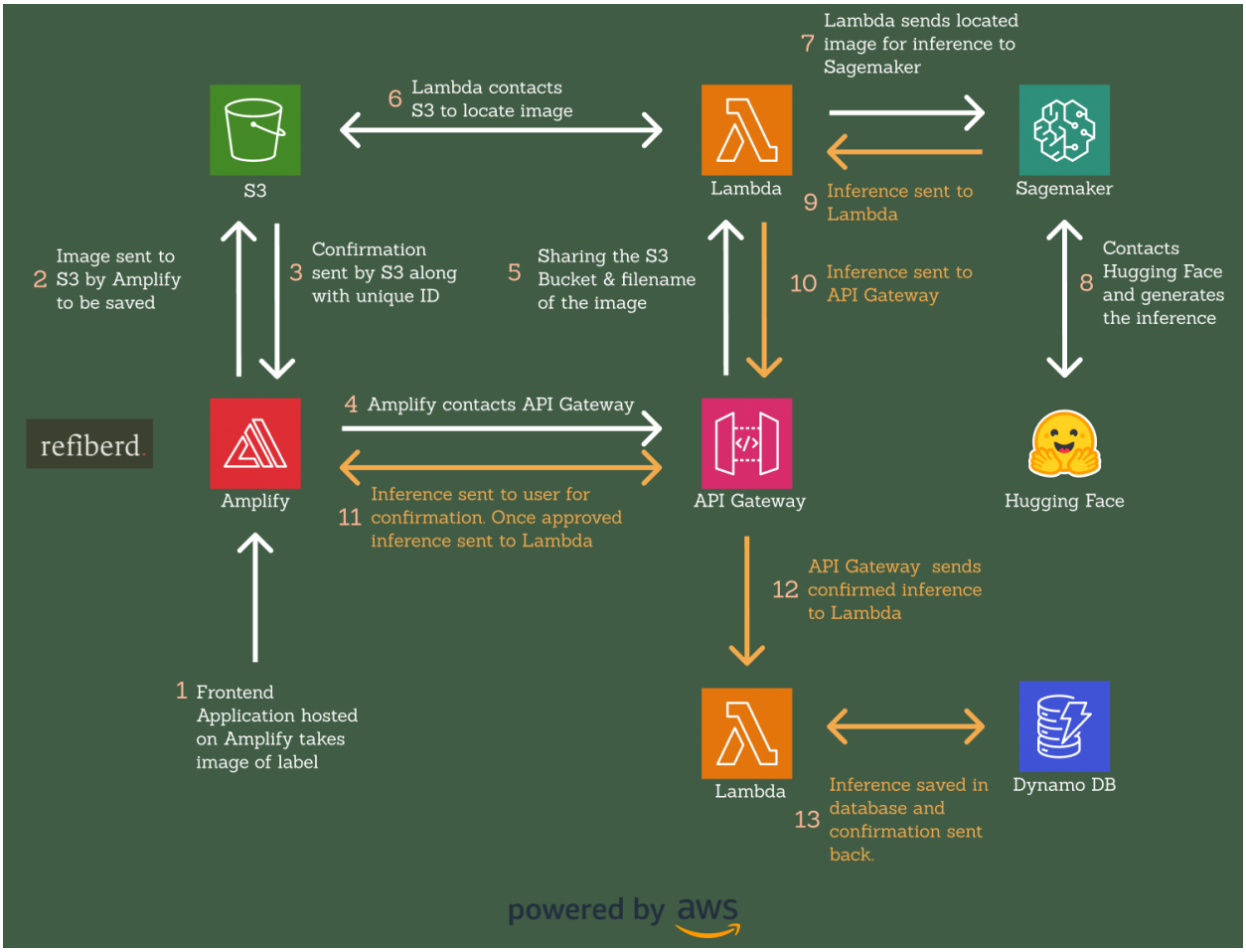

Figure 5 - End-to-End Application Architecture.

Front End and Amplify

Amplify is an AWS service that allows users to build full-stack web and mobile apps. It also allows hosting for both production and development level deployments of the front end (Figure 6). Amplify seamlessly connects with other AWS services and features such as storage, authentication, and APIs, all of which can be instantiated from within the Amplify interface as part of the project. Additionally, while in test, it deploys a temporary webpage link, allowing us to try the web application on our phones and laptops without having the code repository and Amplify configured manually.

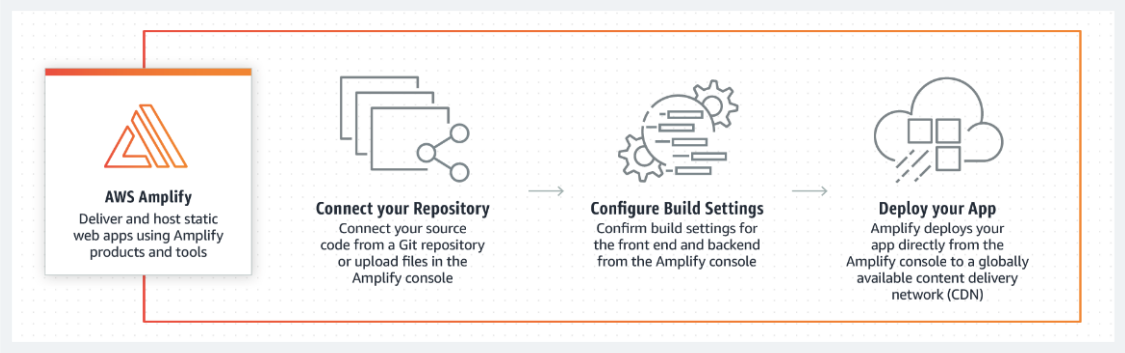

Figure 6 - How hosting with Amplify works.[5]

An essential step when using any AWS service is to set up an IAM Role. This role carries a set of permissions that define what actions are allowed and denied for various AWS resources. After that, all the services that needs to be created for the Amplify project are managed through the Amplify CLI (terminal) and a local IDE for interacting with the react.js project.

The process for creating, updating and deploying the front end on Amplify is as follows:

1. **Initialize the project:** To start, we initialize the Amplify project with the `amplify init` command and authenticate with the IAM Role created for this project. Then, choose a programming language. In our case, we chose React.js. With this, the console automatically creates a basic React project with its default folders and files (similar to creating a project using node project manager).

2. **Develop React.js components:** for this project, we focused on having the most simple and self-explanatory user-interface possible considering that it is an internal tool for Refiberd and the main goal was to reduce time spent on labeling new data. The application was only a portion of our capstone efforts, which also included solutioning, research, and model-fine tuning. As a result, the user experience consists of four simple steps that translate into six components:
    a. Main Page: after authentication, the user arrives to the home page which has a button to take a picture.
    b. Camera Page: this component opens the camera of the user's device, asks for permission to access the camera, takes a picture, saves the image in S3, and calls the model API with the image storage metadata from the S3 bucket.
    c. Loader Page: an intermediate page when the model is working on the prediction showing a throbber. Its objective is to signal to the user that the App is processing the image and prevent them from double clicking and causing an error.
    d. Form Page: renders the model prediction results in a table and allows the user to edit as necessary. After user confirmation, Amplify calls the API that saves the labeled data into DynamoDB.

[5] https://aws.amazon.com/amplify/hosting/

e. Success: signals to the user that the tag composition and image were saved successfully in the database.
f. Header: a component is present in all the previous components as a menu banner. It has the Refiberd logo and a Sign Out button.

3. **Add additional features:** AWS Amplify has additional built-in components that can reduce the coding necessary for the application. We took advantage of these by leveraging Amazon Cognito. This service adds a first authentication page (Sign in and Sign up) that can be easily managed in the AWS Cognito console. The benefit of this service is that allows us and, in the future, Refiberd, to control who is using the application, add and delete users, as well as other configurations such as mode of authentication (username, email, phone) and messages sent to the users (emails).

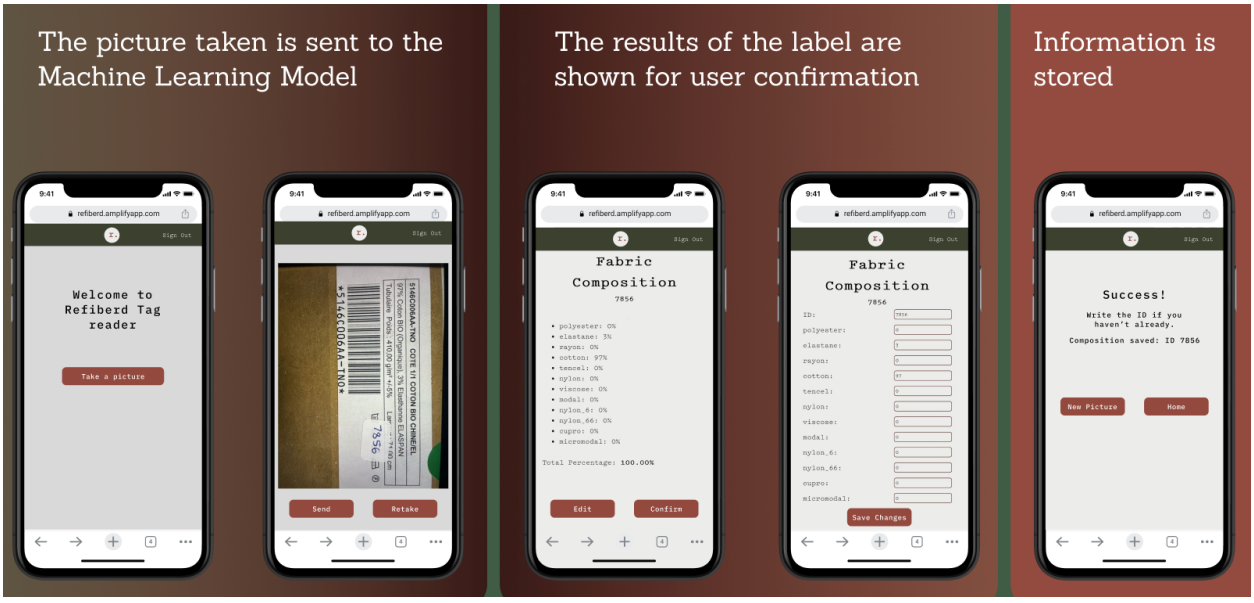

Figure 7 - The front-end user flow.

Back End - Passing Data and Returning Inference

Connecting to a database

We streamlined the connection between the user interface and our backend services using AWS Lambda and API Gateway. The process begins at the AWS Amplify powered front end, where once user confirmation of model results or completion of edits triggers a call to the backend API. These requests are routed through Amazon API Gateway, which act as a managed service making it easier to create, publish, maintain, monitor, and secure APIs at scale.[6]

Once a request is received, API Gateway forwards it to the appropriate AWS Lambda function. Lambda offers a serverless compute service that lets you run code without provisioning or

[6] https://aws.amazon.com/api-gateway/

managing servers.[7] This flexibility is crucial for handling the varying load of API requests efficiently.

The Lambda function processes the request, which often involves interacting with our database. For our data storage needs, we utilize AWS DynamoDB, a fast and flexible NoSQL database service for all applications that need consistent, single-digit millisecond latency at any scale. It's a fully managed database and supports both document and key-value store models.[8]

In our specific use case, each request to the Lambda function involves saving data to DynamoDB. The database is structured to contain a table, with `sample_id` as partition key (primary key), where the table stores the percentages of materials confirmed by our human user. This setup not only allows for quick data storage but also ensures that our data management is scalable and secure, meeting the demands of our application's users effectively. The database also stores the composition information per requirements provided by Refiberd in addition to the location of the photo taken and stored in an S3 bucket for future access and auditing as needed.

sample_id *(String)*	cotton	cupro	elastane	imageKey	micromodal	modal
6504	0	0	3	photo_171...	0	0
6523	100	0	0	photo_171...	0	0
6534	100	0	0	photo_171...	0	0
7856	97	0	3	photo_171...	0	0

Figure 8 - Screenshot of database preview.

By integrating these AWS services, we have created a robust backend architecture that supports the seamless operation of our application, ensuring efficient data flow from the front end all the way through to our database.

Hosting the Model

Custom Inference via Hugging Face and Amazon Sagemaker

Per much research on industry best practices for integrating a machine learning model into the flow of a user application via AWS, we decided to build an inference endpoint on Amazon Sagemaker. Amazon SageMaker is “a fully managed machine learning (ML) service” which can be used to deploy machine learning models in a production environment.[9]

[7] https://aws.amazon.com/lambda/
[8] https://aws.amazon.com/dynamodb/
[9] https://docs.aws.amazon.com/sagemaker/latest/dg/whatis.html

Our model was hosted on Hugging Face as that was where it was fine-tuned. Fortunately, Sagemaker offers a connection to Hugging Face such that a public model can be called with no more than the model URL.[10]

As with much of our AWS journey, what seemed to be simple, quickly became complicated. The AWS documentation proved to often be out of date and sent us searching in circles. Debugging in a fully-managed environment took getting comfortable with the AWS logging console and understanding where and how to place print statements and other tracking indicators to find bugs. Furthermore, deploying our model proved more complicated than simply using the auto-generated AWS deployment script available for our model on Hugging Face.

The way the model is currently deployed leverages use of AWS pre-built, hosted, deep-learning container images[11] that accommodate the Hugging Face transformers and PyTorch libraries needed to run the inference script. Sagemaker supports the deployment of custom inference models via inference script, stored using particular scripting conventions and file structures in Amazon S3 via tar archive. We experienced significant difficulty related to slight differences between documentation regarding overriding the HuggingFaceHandler service[12] and the AWS documentation for hosting a custom inference script[13]. By combining approaches, we were able to create a custom inference script that runs the majority of the pre-processing when passed an image.

The script imports necessary transformers and the PyTorch, which in turn allows us to instantiate both the processor and the model. Images passed to the endpoint in .jpg format are pre-treated, tokenized and sent to the model hosted on Hugging Face for predictions. Predictions are returned in JSON format by the endpoint. The inference script contains model, input, predict and output functions, per AWS requirements. The scripts and a requirements file are compressed into a tar.gz file, which is then stored in an Amazon S3 bucket. The bucket functions much like Google Drive, storing unstructured files and assigning them both a URI and URL for easy access within the AWS environment.

With the custom inference script stored in S3, first the necessary IAM permissions must be set up such that the S3 bucket and Amazon Sagemaker can communicate with each other (i.e. files on S3 can be accessed from within Sagemaker studio). Once permissions are set up, a notebook can be opened in a JupyterLab instance, hosted on Sagemaker studio. Via boto3, Amazon's software development kit for Python, and the connector between Hugging Face and Sagemaker, both the model and the appropriate endpoint can be created from within the notebook.

[10] https://huggingface.co/docs/sagemaker/en/index
[11] https://github.com/aws/deep-learning-containers/blob/master/available_images.md
[12] https://github.com/aws/sagemaker-huggingface-inference-toolkit
[13] https://docs.aws.amazon.com/sagemaker/latest/dg/your-algorithms-inference-main.html

```
[2]: sagemaker = boto3.client('sagemaker')

model_name = "space-donut"
model_url = [redacted]  # S3 URL to your model artifact
role_arn = [redacted]  # Your SageMaker IAM role ARN

container = {
    'Image': '763104351884.dkr.ecr.us-east-1.amazonaws.com/huggingface-pytorch-inference:2.1.0-transformers4.37.0-cpu-py310-ubuntu22.04',
    'ModelDataUrl': model_url
}

create_model_response = sagemaker.create_model(
    ModelName=model_name,
    ExecutionRoleArn=role_arn,
    PrimaryContainer=container
)

print(create_model_response)

{'ModelArn': [redacted], 'ResponseMetadata': {'RequestId': [redacted], 'HTTPStatusCode': 200, 'HTTPHeaders': {'x-amzn-requestid': [redacted], 'content-type': 'application/x-amz-json-1.1', 'content-length': '73', 'date': 'Tue, 23 Apr 2024 15:35:24 GMT'}, 'RetryAttempts': 0}}
```

Figure 9 - Model creation.

```
[3]: endpoint_config_name = "space-donut-endpoint-config"
model_name = "space-donut"  # Ensure this matches the model name used when creating the model

create_endpoint_config_response = sagemaker.create_endpoint_config(
    EndpointConfigName=endpoint_config_name,
    ProductionVariants=[
        {
            'VariantName': 'AllTraffic',
            'ModelName': model_name,
            'InitialInstanceCount': 1,
            'InstanceType': 'ml.m5.large',  # Choose the instance type based on your needs
            'InitialVariantWeight': 1
        },
    ]
)

endpoint_name = "space-donut-endpoint"
create_endpoint_response = sagemaker.create_endpoint(
    EndpointName=endpoint_name,
    EndpointConfigName=endpoint_config_name
)

print(create_endpoint_response)

{'EndpointArn': [redacted], 'ResponseMetadata': {'RequestId': [redacted], 'HTTPStatusCode': 200, 'HTTPHeaders': {'x-amzn-requestid': [redacted], 'content-type': 'application/x-amz-json-1.1', 'content-length': '88', 'date': 'Tue, 23 Apr 2024 15:35:31 GMT'}, 'RetryAttempts': 0}}
```

Figure 10 - Endpoint Creation

Once both the model and endpoint are created, boto3 can be used to instantiate a Sagemaker runtime and invoke the endpoint from a python script, passing in the jpg input to receive an output.

Invoking the Model Endpoint via API

Once the endpoint is created, an API is needed to access the endpoint from the front end. API access to the Sagemaker endpoint allows for control of access to the endpoint, as invocations of the endpoint are charged per request and must be protected with the necessary safety precautions.

Again, the development of this pipeline proved more difficult than expected, as documentation often proved out-of-date.[14] AWS hosts a service called API gateway which allows for the easy creation of API endpoints, which can be directly integrated with a Lambda function on AWS Lambda. Lambda offers a method of hosting a script in any language that runs in response to

[14] https://aws.amazon.com/blogs/machine-learning/call-an-amazon-sagemaker-model-endpoint-using-amazon-api-gateway-and-aws-lambda/

an action, as opposed to being hosted and incurring cost via EC2 instance.[15] The user interface for

Lambda initially made this appear to be incredibly simple and low-code, with the only code written corresponding to the Lambda script that invoked the Sagemaker endpoint. Necessary IAM access had to be set up such that the front end could hit the API Gateway, which could in turn access the Lambda function and the Lambda function could access Sagemaker accordingly. The issue came as the JPG files being sent as a blob proved to be too large. The images themselves were under the 6MB payload limit imposed by Lambda, however API Gateway turns the images into byte strings and passes them with headers per cross-origin resource sharing (CORS)[16] and HTTP protocol, which quickly inflated the size of the binary encoded image.

Fortunately, many-a-developer has faced similar challenges and we quickly found workarounds to our problem. If an image is hosted in S3, as opposed to being passed via payload, the payload limit can be bypassed, as the Lambda function simply accesses the JPG image directly using its unique S3 resource address.[17] While Lambda enforces data processing limits, the size of the image being pulled from S3 was well below that limit and the payload would solely contain the name of the S3 bucket and the desired image address.

This change in architecture proved necessary, however it also serendipitously allowed us to store the images being taken on the front end to allow for future spot checks and audits. The front-end architecture was modified to pass the image to an S3 bucket, assigning it an address based on a unique ID generated according to the timestamp at which it was taken. The front end then awaits confirmation of storage, at which point it calls the API passing in the image address and S3 bucket name. This information travels through the API hosted on API gateway, activating the Lambda function, which plugs it into the Sagemaker endpoint. The endpoint passes back a JSON, which is packaged within the appropriate headers to meet CORS protocol.[18] This JSON payload makes its way through the API gateway and returns as a response within the Amplify front end.

The inference takes a few seconds, which is a source of latency in the application. A throbber was added to the user interface to prevent the user from clicking multiple times while inference is occuring, as this causes an error.

Learnings from Hosting the Model

If we were to go back in time and design our architecture from the ground up, it could have been wise to fine-tune the model within the Sagemaker environment, for the purposes of ease of

[15] https://aws.amazon.com/lambda/
[16] https://docs.aws.amazon.com/AmazonS3/latest/userguide/cors.html
[17] https://theburningmonk.com/2020/04/hit-the-6mb-lambda-payload-limit-heres-what-you-can-do/#:~:text=AWS%20Lambda%20has%20a%206MB,his%20cat%20to%20your%20app.
[18] https://docs.aws.amazon.com/lambda/latest/api/API_Cors.html

integration. However, Sagemaker was by far the most expensive service used by our team, with the use of their Sagemaker Studio Domain, from which JupyterHub can be accessed, constituting the bulk of our cost. Hugging Face allowed us to fine-tune our model for free and did not incur any cost during the process.

Though Sagemaker is expensive, it is incredibly powerful and backed by a wide array of docker containers and images which allow flexibility and relative ease when compared to hosting a machine learning model locally.

If we had additional time, it would likely be in our interest to better understand methods to reduce latency at the point of inference.

Reality of Deployment + Recommendations

Pricing

Given our current workflow, where users let the AI make a prediction and then confirm or edit the predictions, and considering the possibility of a new workflow that allows for batch processing of multiple photos, we need to explore cost optimization strategies that enhance both setups. Here's a more comprehensive approach that includes optimizations for our existing workflow as well as potential changes:

Optimizing Costs in Current Workflow

- **Right-Size SageMaker Instances:** We must ensure that the instances powering our SageMaker models are optimally configured. This involves selecting instance types that accurately meet our computational needs without over-provisioning, thus avoiding excessive costs. Regular reviews of instance performance and cost will help us determine if we can switch to a more cost-effective instance type without compromising on performance.

- **Model Optimization:** Enhancing the efficiency of our model can lead to quicker predictions and reduced computational demands. This might involve simplifying the model architecture, optimizing the inference code, or implementing techniques like model pruning or quantization. Efficient models use less compute time per prediction, directly translating into lower costs.

- **Cost Monitoring and Alerts:** Implement detailed monitoring with tools like AWS Cost Explorer to keep a continuous check on where costs are being incurred and identify potential savings. Set up budget alerts to proactively manage costs and avoid surprises in your AWS bill.

- **Proposed Workflow Change:** We can adjust our workflow to allow users to take multiple photos, submit them simultaneously, and then receive multiple predictions through batch processing. This approach would enable users to confirm the accuracy of each prediction one by one after all predictions have been made. This change would leverage batch

processing, which can be more cost-efficient compared to processing each request individually in real-time.

Benefits of the New Workflow

- **Batch Processing**: By grouping multiple prediction requests together, we can make more efficient use of our computational resources. Batch processing reduces the overhead of starting up and tearing down processes for individual predictions, thus lowering costs. We can schedule these batch processes during off-peak hours or when spot instance prices are lower, maximizing cost savings.

- **Utilization of Spot Instances**[19]: With batch processing in place, we can utilize Spot Instances to handle our prediction workloads. Since batch processes can tolerate some interruption and do not require immediate real-time processing, Spot Instances become a viable option. Spot instance enables 90% cost saving compared to on-demand services. Spot Instances are less reliable in terms of availability, but since our batch processing does not demand immediate execution, we can design our system to handle interruptions gracefully.

Possible security concerns

Since the web application is hosted in Amplify, there are many configurations for security that can be configured through AWS console. Overall, however, cross-origin resource sharing protocol and the centralized nature of AWS makes ensuring security easy for the development team. As long as the IAM and ARN IDs are not exposed in any part of the application, the SOC2 certification and AWS's rigorous security measures largely outsource security beyond a handful of parameters defined by the user. The architecture is technically bound by Amazon's shared responsibility model[20], so the following represent areas we had to configure and consider throughout the development process:

- **Identity and Access Management:** The Amplify owner controls the policies of who can access the application and which services can communicate internally. Policies are attached to users which can be associated with developers or the services themselves. AWS evaluates the policies when a principal (user, root user, or role session) makes a request and is responsible for ensuring this process of verification in their architecture is secure. All the permissions policies are stored in AWS in JSON format. These policies and roles had to be set up and configured every step of the way and we followed the principle of least privilege[21] to ensure roles were only given the permissions necessary to enable the application to function. It is also critical that these IDs and ARN numbers do

[19] https://aws.amazon.com/ec2/spot/
[20] https://aws.amazon.com/compliance/shared-responsibility-model/
[21] https://csrc.nist.gov/glossary/term/least_privilege

not exist in public format anywhere, otherwise the permissions associated with the ID can be used by anyone with access.

- **Data Protection:** AWS is responsible for protecting the global infrastructure and maintains control over data hosted on it. In our case, we are using Encryption at Rest, that is managed by Amazon CloudFront. This service uses SSDs encryption.[22]
- **Security best practices:** at the moment of this report, the web application is hosted in the default amplifyapp.com domain. When delivering to Refiberd, we are going to transfer it to Refiberd domain and use cookies with a `__Host-` prefix as recommended.[23]
- **Image Encryption**: The images of the labels represent ground truth, but do not represent the intellectual property of the client, as they are not hyperspectral images of the fabric swatches themselves. As a result, this data is not encrypted prior to being sent to S3.
- **Docker Images:** There are minor concerns related to the use of a publicly hosted deep learning container, however this architecture is protected by shared responsibility on the part of AWS.[24]
- **Cross-Origin Resource Sharing Enablement:** For the APIs built on API Gateway, cross-origin resource sharing is enabled as a method of increasing security when data is being passed across services existing on different services within AWS.

The Model

In traditional image-to-text captioning approaches, the typical workflow involves utilizing off-the-shelf Optical Character Recognition (OCR) engines to extract text, followed by the application of natural language processing techniques. These techniques often range from regular expressions to decoder-only large language models like GPT, which predict the subsequent tokens. However, in our study, we opted for a more innovative approach by adopting and fine-tuning the OCR-Free Document Understanding Transformer, a pioneering method introduced by kim2022 et al.[25] This model stands out by employing a transformer-only architecture, which notably reduces computation costs due to its reliance on a relatively modest 220M parameter model. As depicted in Figure 6, the Donut model incorporates a Swin Transformer as its backbone to serve as the encoder, paired with a BART decoder. It's important to emphasize that this figure primarily illustrates the structural framework of the model; we encourage readers to consult the original publication for a comprehensive understanding of its full capabilities.

In alignment with our focus on document analysis, we utilized a pretrained model available on Hugging Face and proceeded to fine-tune this model on a self collected dataset. This dataset was collected at Refiberd's facility and preprocessed as described below.

[22] https://docs.aws.amazon.com/amplify/latest/userguide/encryption-at-rest.html
[23] https://docs.aws.amazon.com/amplify/latest/userguide/security-best-practices.html
[24] https://docs.aws.amazon.com/deep-learning-containers/latest/devguide/security.html
[25] https://doi.org/10.48550/arXiv.2111.15664

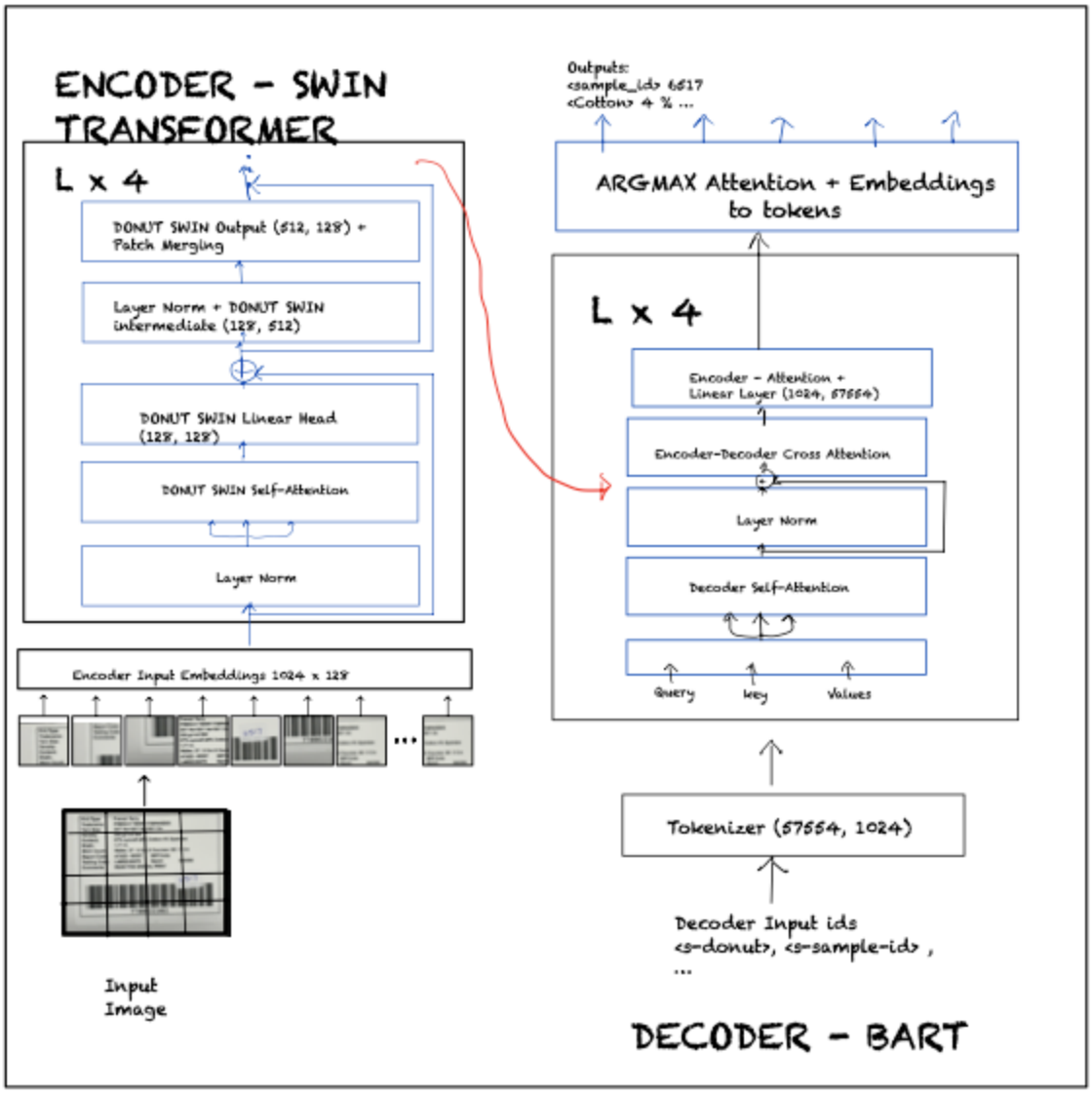

Figure 11 - Model Architecture and backbone.

Data Preprocessing:

Since our approach relied on fine-tuning a multi-modal transformer model with no reliance on a conventional OCR engine, our goal was to collect the highest resolution dataset at source and experiment with multiple preprocessing techniques to achieve the optimal model performance. Thus, we collected the images in raw form which accounted for a 20 GB dataset. However, due to compute constraints, both in terms of system ram and GPU compute, we downsampled the images from 4032x3024 pixels to 1280x960 pixels. Additionally, since we were leveraging the Hugging Face hub for downloading the model and its processor including tokenizer, we converted the raw dataset into .jpeg form and created a HuggingFace Image Dataset object [26] that uses the Apache Arrow format to store data in a columnar memory layout while ensuring efficient memory

[26] https://huggingface.co/docs/hub/en/datasets-image

mapping. Additionally, we uploaded this dataset on the HuggingFace hub (as a private dataset) to be shared with Refiberd's team for future purposes. Our dataset split is shown as follows:

Train Set	469
Validation Set	119
Test Set	66

Table 2 - Dataset split.

Model Setup:

The initial step in preparing our pipeline was to load both the model (with pre-trained checkpoints) and the corresponding processor including tokenizer from the Hugging Face hub. Next, we adjusted the image dimensions to 1280x960 which required altering the encoder image size configuration. Furthermore, the token sequence length was also increased from 20 to 768. The process of fine-tuning drew heavily from the steps and methodologies outlined in the documentation released by Hugging Face [27]. To utilize the model's image-to-text captioning capabilities, the images were then converted into pixel values using the encoder processor while including random padding to bolster the model's robustness during inference time. Additionally, since our fine-tuning paradigm required next-word token prediction, we incorporated the fabric composition types such as cotton, nylon, polyester etc. as special tokens thereby increasing the decoder's vocabulary by an additional 30 tokens. This step proved pivotal in obtaining the desired accuracy downstream as we didn't want the model to split these words into sub-tokens thereby rendering the process of predicting fabric compositions for these tokens futile. Complete details regarding this setup and architecture, including all adjustments and configurations, are documented and accessed through our GitHub repository link [28]. Additionally, we believe that this comprehensive resource which includes a custom defined PyTorch training loop might serve as a valuable resource for replicating and/or extending our project on other self-curated datasets out in the wild.

Training Loop:

Given the constraints on our compute power, we were unable to experiment with a comprehensive grid search approach to experiment with various hyper-parameter settings. Instead, we chose a handful of parameter configurations based on assessing the preliminary training results and assessed the training losses and validation accuracies per epoch. Primarily, we experimented with varying choices of learning rates and optimizers which are displayed in Table 1 below. Ideally, we wanted to experiment with varying batch sizes but were unable to train the model given our compute constraints, so we kept the batch size to 1.

[27] https://huggingface.co/docs/transformers/en/model_doc/donut
[28] https://github.com/azhara001/Fabric-Composition-Extraction

Our experimental setup thus included six distinct configurations, each of which was run on an NVIDIA L4 Tensor Core GPU equipped with 22 GB of RAM with an average training duration of 75 minutes per configuration for five epochs with validation accuracy measured per epoch.

Configuration	Learning Rate	Optimizer
1	3e-5	Stochastic Gradient Descent
2	3e-5	Stochastic Gradient Descent with Momentum
3	3e-5	Adam
4	3e-5	AdamW
5	3e-4	Adam
6	1e-5	Adam

Table 3 - Hyperparameters Explored

Loss Function and Evaluation Metric

As the decoder architecture relies on next token prediction, we used the cross-entropy loss as our primary training metric. This involved extracting the output logits from the model's last hidden states, with each token represented by a dimension of (1,768). These logits were then fed into a linear head culminating in an output size of (768, 56000) where 56000 represents the approximate size of the tokenizer's vocabulary. Following this, we computed the argmax to identify and predict the token with the highest probability. This method proved effective, as evident from the significant and almost immediate reduction in training loss when using Adam and AdamW was our choice of optimizers in Figure 7. Overall, our loss curves regarding our choices of optimizers are intuitive where we observe that SGD does start to converge but takes a lot more time than SGD with momentum whereas the Adaptive choice of optimizers like Adam and AdamW converge to the same minima after 5 epochs. We also experimented with varying choices of learning rates and observed the model diverging after 1200 epochs for a higher learning rate of 3e-4. Additionally, we were interested in exploring the potential of using cosine similarity loss which would have involved the conversion of ground truth tokens into embeddings. However, due to our compute constraints, we were unable to pursue experiments with alternative loss functions - a direction that we would wish to work on in the future.

To effectively measure the performance of our model on both the validation and test sets, we sought an evaluation metric that could accurately reflect the precision required int he predicted sequence of tokens. Given our goal of achieving a 100% match in the next token sequence prediction, we chose to use the Levenshtein Distance as our primary metric. The Levenshtein distance quantifies the number of single-character edits (insertions, deletions, substitutions) required to change the predicted sequence into the ground truth label. Additionally, we

normalized this distance by the token length such that a value of 0 indicates a perfect match whereas an upper bound of 1 indicates no match at all. We can clearly see a direct relationship between the training loss and levenshtein score where SGD performs poorly during the validation stage.

This approach to evaluation is reflected in our plotted validation accuracy shown in Figure 7, where the model demonstrates impressive performance. Although evaluating large language models (LLMs) remains an active area of research, utilizing the Levenshtein distance provided a robust starting point for our analyses.

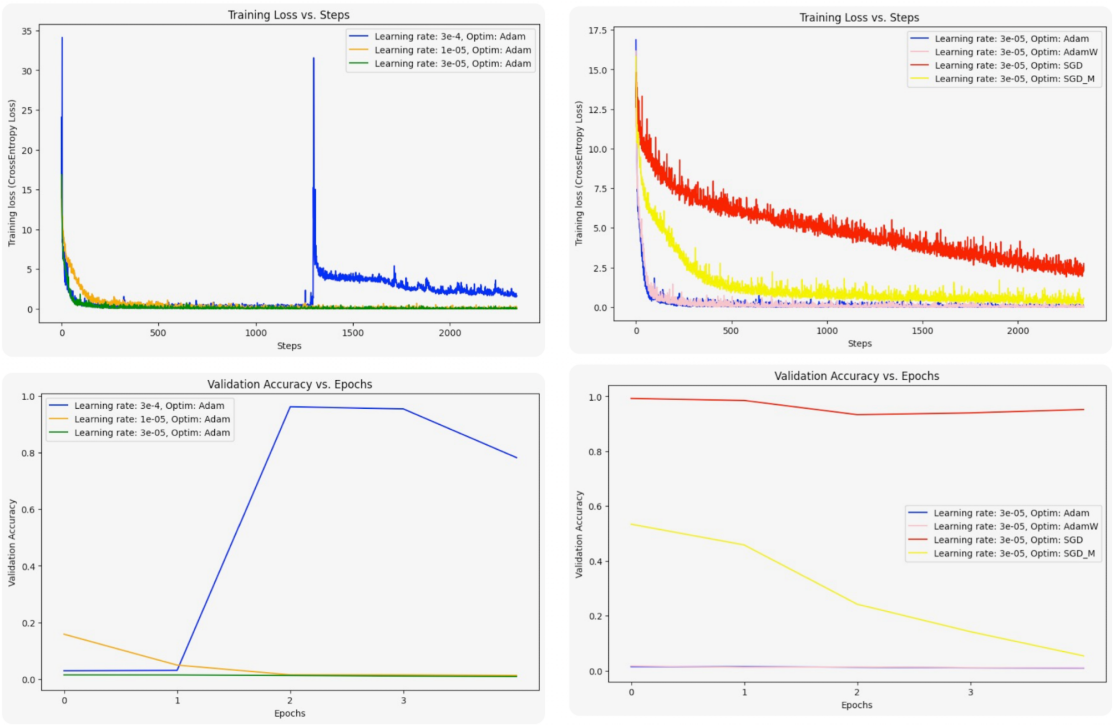

Figure 12 - Training Loop Results

Model Performance on Test Set

Following our fine-tuning paradigm on the multiple hyper-parameter settings as defined in table 1, we chose a learning rate of 3e-5 with Adam as the optimizer to deploy the model on AWS SageMaker. Additionally, we also assessed the model performance on the test set and achieved impressive results given that our model was only fine-tuned on 569 train images only. The following table summarizes our model performance with inference performed on the final state of the trained model after five epochs for each of the four optimizer choices as hyperparameters.

Optimizer	Samples with perfect match (Total Samples: 66)	Average Normalized Levenshtein Score (0-1)

Stochastic Gradient Descent (SGD)	0	0.95
SGD with Momentum	13	0.03
Adam	40	0.007
AdamW	41	0.007

Table 4 - Inference results on test set.

While observing the results in Table 4, we decided to analyze the performance of the models on a sample from the test set and observe the model predictions for all the four models mentioned above. Figure-8 below shows the image as well as the predicted labels for each model. By observing the model prediction for SGD optimizer, we observe that the decoder is unable to capture the structure of the prediction sequence altogether which is reflected in the high Levenshtein distance. For SGD with momentum, we observe that the model is not only able to get the structure right but is also able to predict the sample id correctly. Adam gets the sample id and the cotton composition correct but is not able to predict the percentage of polyester (which is misspelled). Lastly, with AdamW, we observe a 100% match implying the model got all the sequence and compositions right. It should be noted that the misspelled fabric composition titles render this problem difficult to solve with a conventional OCR engine-based model since that would require fine-tuning a decoder LLM to predict next token based on these misspelled tags which is where our model is able to churn out impressive results.

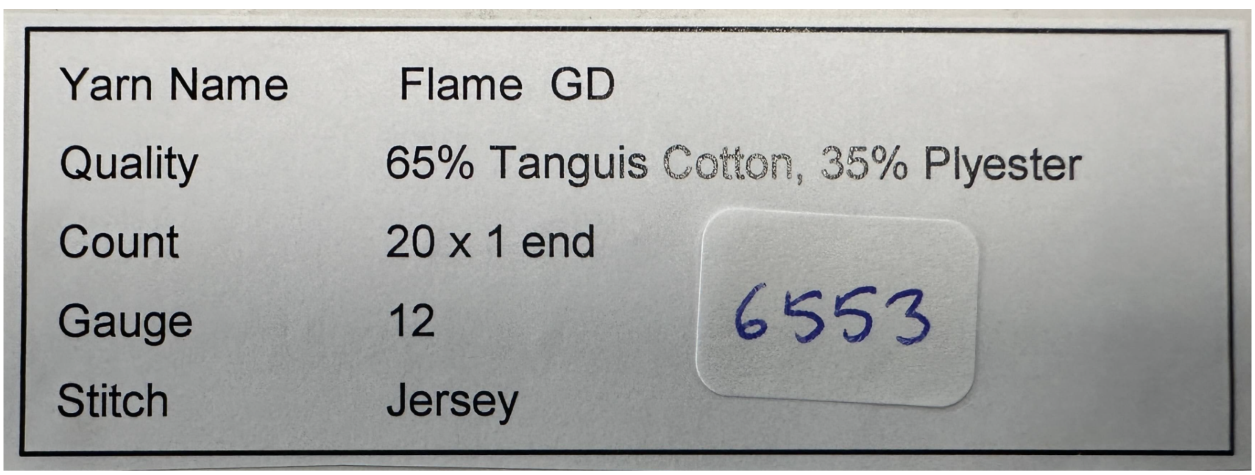

Figure 13 - Sample image for inference (see Table 5)

Model Optimizer	Levenshtein Distance	Model Prediction
SGD	0.93	'text_sequence': ' No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No

		No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No No 08 08 08 08 08 08 08 08 08 08 08 08 08 08 08 08 08 08 6553 6503 65 65 6503 65 65 65 6503 65 65 65 6503 65.00.000000000.00000'}
SGD with momentum	0.02	{'sample_id': '6553.0', 'composition': {'polyester': 'None', 'elastane': 'None', 'rayon': 'None', 'cotton': 'None', 'tencel': 'None', 'nylon': 'None', 'viscose': 'None', 'modal': 'None', 'nylon_6': 'None', 'nylon_66': 'None', 'cupro': 'None', 'micromodal': 'None'}}
Adam	0.01	{'sample_id': '6553.0', 'composition': {'polyester': 'None', 'elastane': 'None', 'rayon': 'None', 'cotton': '65.0', 'tencel': 'None', 'nylon': 'None', 'viscose': 'None', 'modal': 'None', 'nylon_6': 'None', 'nylon_66': 'None', 'cupro': 'None', 'micromodal': 'None'}}
AdamW	0.00	{'sample_id': '6553.0', 'composition': {'polyester': '35.0', 'elastane': 'None', 'rayon': 'None', 'cotton': '65.0', 'tencel': 'None', 'nylon': 'None', 'viscose': 'None', 'modal': 'None', 'nylon_6': 'None', 'nylon_66': 'None', 'cupro': 'None', 'micromodal': 'None'}}

Table 5 - Inference result on sample image (in Figure 13)

VI. Closing remarks

Throughout this project, our team learned new skills and refined knowledge from our studies at the School of Information. This project was a big challenge, not only because of the technical complexities, but also because we started very late as we shifted from a research paper to deliver a product for a company, only beginning on the application mid-February. As a result, however, we all have had the opportunity to work on an end-to-end cloud-based application that leverages machine learning which is an extremely valuable learning result.

AWS is a powerful platform that can help organizations easily manage different digital functionalities for their business, both with the client and internally. However, its complexities are non-trivial, and with all the available services and continuous improvements of the platform it was a challenge to understand what was important for our project and how to use it. Furthermore, the constant updates to AWS have resulted in a rat's nest of documentation that at times felt like chasing a wild goose when we were debugging or problem solving. Fortunately, our team remained flexible and positive and as a result we learned tremendously from our experience. AWS is a widely-used tool and as such this experience provided us with many transferrable skills.

Most importantly, we are delivering a product to a company that will help them reduce the time and headcount needed to label data correctly, expediting the time needed to improve model accuracy, which in-turn will impact deployment and adoption. The consensus across many climate technology enthusiasts and computational sustainability experts is that machine learning and information technologies will not be the star of the show when it comes to solving climate change. However, it will play an important role in expediting the availability of solution, winning an Oscar for best supporting actor in this context. Though the Refiberd tag reader is not directly related to solving the climate change challenge, it is a solution that will help Refiberd to be more efficient in the data gathering process so they can train their machine learning model quickly and effectively. It will help Refiberd solve the textile end-of-life problem - *faster.*