The report should be brief and give me an understanding of the following:

• Which significant choices you have had to make during development, what the options were, and why you made them.

We decided to use pandoc for the process of turning markdown files to pdf. It gave us the advantage that we could include picture like the NTNU logo and a signature. There is a lot of ways to turn md to pdf such pdfkit but, we found it the easiest to use pandoc. The drawback with pandoc is that it's slightly slower than the other processes.

• How the system works and how it is structured.

The project is structured in 2 parts we have the front end and the back end. The front end handles a basic website that allows you to upload a file that file is then send to the backend which processes the file before sending it back to the front end allowing the user to download it as a tar.gz file. The backend takes a CSV file gets out the information of the different names in the CSV file and places them in a template markdown file. The markdown file is then further processed to a pdf files. We decided on using pandoc to turn the md files to pdf. The reason is because using this method allows us to have images on the PDF files. The only issue with this method is that it's a little bit slower than other way of converting markdown files to pdf.

• An overview of the code (regular documentation should be in code).

Backend part of the code:

1. Filling Templates: The script starts by filling a template with data from each row in some data source (not shown in the excerpt). The template is filled by replacing placeholders ({{FirstName}} and {{LastName}}) with actual data from the row. The filled template is then written to a Markdown file. The filename of the Markdown file is derived from the first and last names in the row.
   3
2. Converting Markdown to PDF: After all the Markdown files have been created, the script goes through each Markdown file in the MD_FOLDER directory. It reads the content of the Markdown file and then uses pandoc to convert the Markdown file to a PDF file. The PDF file is saved in the PDF_FOLDER directory with the same name as the Markdown file, but with a .pdf extension instead of .md.

3. Error Handling: If the pandoc command returns a non-zero exit status (which indicates an error), the script prints an error message and returns a 500 status code.

4. Compressing PDFs: After all the Markdown files have been converted to PDF, the script compresses all the PDF files into a tar.gz file named PDF.tar.gz. However, the code for this part is not shown in the excerpt.

Frontend part of the code:

1. Directory Setup: The script starts by defining three directory paths: UPLOAD_DIR, COMPRESSED_DIR, and DOWNLOAD_DIR. These directories are then created using os.makedirs, with exist_ok=True to avoid an error if the directories already exist. A path for the compressed file (compressed_files.tar.gz) is also defined.

2. Compress CSV Files Function: The compress_csv_files function is defined to compress all CSV files in a given folder into a tar.gz file. It takes two arguments: folder_path (the directory containing the CSV files to compress) and compressed_file_path (the path where the compressed file will be saved). It uses the tarfile module to create a new tar.gz file and add all the CSV files from the specified folder.

3. File Upload Endpoint: The @app.post("/uploadfile/") decorator defines a new endpoint in a FastAPI application for uploading files. The create_upload_files function is called when a POST request is made to this endpoint. It takes a list of UploadFile objects as an argument, which represent the files being uploaded. The function then iterates over the directories (UPLOAD_DIR, COMPRESSED_DIR, DOWNLOAD_DIR) and lists the files in each directory. However, the code provided doesn't show what happens with these files.

For the upload python script

1. Check if the File Exists: The script starts by checking if the file to be uploaded exists on the local system using os.path.exists(file_path).

2. Upload the File: If the file exists, it is opened in binary read mode ('rb'). A dictionary is created with a single key-value pair. The key is 'file' and the value is a tuple containing the file name and the file object. This dictionary is then passed to requests.post(upload_url, files=files) to upload the file to the server. The server's response is printed to the console.

3. Wait for Server Processing: The script pauses for 5 seconds using time.sleep(5). This is to give the server time to process the uploaded file.

4. Download the File: After the pause, the script sends a GET request to download_url to download a file from the server. If the server responds with a 200-status code (indicating success), the script writes the content of the response to a file on the local system. If the server responds with any other status code, the script prints an error message.

• How to use the system.

Installation: Clone the repository to your local machine using git clone. Navigate to the project directory using cd IDG2001--ASSIGNMENT1. Install the necessary dependencies using pip install -r requirements.txt.

Building the Docker Image: Build the Docker image for the backend server using the provided Dockerfile. Run the command docker build -t your-project-name . in the terminal, replacing your-project-name with the name of your project.

Running the System Locally: Run the main.py file to start the server. You can access the endpoints through the browser or a tool like Postman. You also need to run the backend server, where the file uploads are processed before you can download the file again. Larger files take longer to process.

Accessing the Hosted System: The frontend and backend servers are hosted on Railway. You can access the frontend at https://oblig1-cloud-frontend-production.up.railway.app/ and the backend at https://quickest-hair-production.up.railway.app/.

• How to set up a system like this.

Work on small parts at time. Set up the api that can handle files and process them. Set up the frontend that can send a request to the api so the api can return files to the frontend.

• How to replace the file processing with another use case.

To replace the file processing, you should go through these steps:

Identify the New Use Case: Determine what you want the system to do instead of processing files. This could be anything from handling user authentication to managing a database of products.

Modify the Back-End Code: Update the back-end code to handle the new use case. This might involve writing new functions or modifying existing ones. For example, if your new use case is user authentication, you might need to write functions to register new users, log in existing users, and check if a user is logged in.

Update the API Endpoints: Modify the API endpoints in the back-end code to handle requests related to the new use case. For example, you might need to create new endpoints for registering and logging in users.

Modify the Front-End Code: Update the front-end code to send requests to the new API endpoints. This might involve writing new functions or modifying existing ones. For example, you might need to write functions to send a registration or login request when a user fills out a form.

Test the System: After you've made these changes, test the system to make sure the new use case is handled correctly. This might involve writing new unit tests or manually testing the system.