FILEFLO

Technical Guide

Students

Aaron Cleary - 19495324

Joao Pereira - 19354106

Supervisor

Geoffrey Hamilton

Date Completed

06/05/23

Table of Contents

1. Abstract

Fileflo is a user-friendly file-sharing platform designed to provide a secure, decentralised alternative to traditional file-sharing services. At its core, Fileflo leverages the InterPlanetary File System (IPFS) to enable peer-to-peer file-sharing across a network of distributed nodes. By utilising IPFS, Fileflo ensures that files are not stored on a single centralised server, but rather across a network of nodes, making them highly available and resilient. In addition to the distributed storage of files, Fileflo also prioritises user privacy and data security by encrypting files with both symmetric and asymmetric encryption algorithms, ensuring that only the owner of the file, or those they choose to share it with, have access to its contents.

To ensure that files can be easily found and downloaded, Fileflo uploads the metadata of each file to an Ethereum blockchain. This metadata includes the file's IPFS hash and other relevant information, which is then used during the download process to locate the file on the IPFS network and download it. By storing this metadata on the blockchain, Fileflo can guarantee that each file's provenance and history can be traced back to its original owner, and that any attempts to modify or delete the file will be recorded on the blockchain. Fileflo also features an intuitive user interface that allows users to easily upload, share, and download files. Users can also create shared folders and invite others to join, enabling seamless collaboration across teams and groups.

2. Motivation

The motivation behind Fileflo lies in the need for a secure, decentralised, and private alternative to traditional file-sharing services. With the rise of cloud-based file storage and sharing platforms, many users have rightfully grown increasingly concerned about the security of their data, as well as the potential for data breaches and cyber attacks. Moreover, centralised file-sharing services often rely on a single point of failure, leaving them vulnerable to disruption and data loss. Fileflo aims to address these concerns by leveraging the power of decentralised technologies, such as IPFS and blockchain, to create a more secure and resilient platform for file sharing.

The privacy of users has also become a growing concern in recent years, with people becoming more aware of the ways in which their data is being collected, used, and shared. This has led to a demand for platforms that prioritise user privacy and data security. By allowing users to encrypt their files before uploading them, Fileflo ensures that only the file owner or those they choose to share it with have access to the contents of the file. This approach to file-sharing ensures that users have complete control over their data and can share files with peace of mind.

Another key motivation behind Fileflo's creation is the desire for a more equitable and inclusive internet. Centralised file storage and sharing systems often lead to the concentration of power and control in the hands of a few entities, resulting in potential censorship, data manipulation, and restricted access. Fileflo's decentralised nature redistributes control over content and data to the users, ensuring no single entity holds disproportionate influence.

Lastly, environmental sustainability has become an increasingly crucial consideration in the development of new technologies. Traditional data centres consume vast amounts of energy and contribute significantly to carbon emissions. By employing a distributed file storage system, Fileflo helps to reduce the environmental impact of data storage and sharing, aligning with global efforts to minimise the carbon footprint of the digital world.

3. Research

3.1 IPFS

Researching IPFS was critical for our project because we needed to find an alternative to our original idea of storing files on the blockchain, which we quickly discovered was far too inefficient for an application such as ours. Storing files directly on the blockchain would have required an immense amount of computing power, storage space, and network bandwidth, making it ridiculously expensive and slow. Through researching other strategies, we discovered IPFS, which is a distributed network protocol that provided an elegant solution to our problem. We learned how IPFS uses a unique content-addressed file system that identifies files by their content rather than their location, making them highly available and resilient. We also learned how IPFS allows users to quickly and easily retrieve files by using the unique cryptographic hash that identifies each file. This ensured that users could access their files quickly and efficiently, regardless of their location or the number of times the file had been accessed previously.

3.2 Blockchain

It was inevitable that we would need to research blockchain technology because neither of us had any experience with using it, and we had always intended to make it a key focus of our project. After abandoning our original idea of storing files directly on the blockchain, we needed to find a way to incorporate it into Fileflo in a meaningful way. We discovered that blockchain technology could provide a more efficient and cost-effective way to store file metadata, which is essential for facilitating file retrieval and ensuring secure access control. We learned that by using a smart contract on the Ethereum blockchain to store file metadata, we could create a tamper-proof record of each transaction, enabling us to track the provenance and history of each file. The smart contract would store the cryptographic hash of each file on IPFS, along with other relevant information such as the filename and file type. This metadata would be securely stored and verified by the Ethereum blockchain, making it highly resistant to tampering and providing an additional layer of security and transparency to the file-sharing process. This approach also enabled us to use the metadata stored on the blockchain to quickly and easily retrieve the corresponding file on the IPFS network.

3.3 Cryptography

We needed to research cryptography because we wanted to ensure that all files uploaded to our platform were securely encrypted, in order to provide maximum security for our users and safeguard against the public nature of IPFS. We learned that we could employ

symmetric encryption by using a single passphrase to both encrypt and decrypt a file, which we recognised would allow us to efficiently encrypt files with minimal costs to usability. As such, we decided to use the symmetric encryption algorithm AES to encrypt files before they were uploaded to our platform. We also learned about asymmetric encryption, which involves using a pair of keys - a public key and a private key - to encrypt and decrypt data. We decided to use asymmetric encryption to encrypt files and folders that are intended to be shared between two or more users, as this approach ensured that the file or folder could only be accessed by authorised users who possessed the private keys that corresponded to the public key.

3.4 AWS Services

In our project last year, we found AWS services to be highly beneficial and useful, leading us to choose AWS as a serverless backend for Fileflo. To determine the most suitable AWS services for Fileflo, we conducted thorough research and identified a combination of services that would offer a secure, scalable, and easily integrable solution. Key services we utilised include:

- **Amazon Cognito, AWS Amplify,** and **AWS Simple Email Service** for user authentication, allowing us to create custom authentication flows and control user access through the application API.

- **AWS Lambda** and **API Gateway** for a serverless architecture that can easily scale with user demands, providing a second layer of authentication and enabling the creation of custom APIs for user attribute control.

- **Amazon S3** for reliable and scalable file storage and serving, ensuring our application remains available to users when needed.

- **AWS CloudFront** to convert S3's HTTP web deployment to HTTPS, enhancing security and protection for our application and its network.
- **AWS CloudWatch** to detect errors and request defects in order to modify the architecture to prevent these errors from occuring.

For our backend database, we opted for **Amazon DynamoDB**, a scalable and flexible NoSQL database solution. We interconnected DynamoDB with several services, such as:

- **AWS Hardware Security Modules (HSM)** to symmetrically encrypt sensitive data like private keys in the database.
- **AWS AppSync** for enabling GraphQL schema operations between the backend and the database.
- **AWS Key Management Service** to provide the required symmetric keys.

These AWS components form a cohesive architecture, with efficient communication enabled through the permissions and policies we established for the Fileflo IAM role.

3.5 User Interface Design

The user interface was of paramount importance for our project, as we needed to create an intuitive and user-friendly interface that would enable users to easily create and share folders, upload and share files, as well as download files. We also needed to provide users with the ability to view their uploads, search for them, and access them quickly and efficiently. Additionally, we wanted our interface to have a clean and modern CSS design that would be aesthetically pleasing to users. Through our research, we learned about various user interface design principles and techniques, such as user-centred design, information architecture, and UX design. We also explored different front-end technologies, such as HTML, CSS, and JavaScript, that could be used to create an engaging and interactive user interface.

To design the user interface for Fileflo, we focused on a user-centred design approach that prioritised the needs and preferences of our users. We conducted user research and usability testing to gain insights into user behaviour and preferences, which we used to inform our design decisions. We also developed an information architecture that organised content and functionality in a logical and intuitive manner. We created a clear navigation structure that enabled users to easily access different sections of the platform, particularly their uploads and folders. For the CSS design of the interface, we focused on creating a clean and modern design that would be visually appealing to users. We researched various different templates until we settled on a minimalist colour palette that matched our logo, clear typography and intuitive icons, which all combined to create a cohesive and aesthetically pleasing design.

4. Design

4.1 System Architecture

The system architecture diagram below showcases the interaction and communication among all components in our technology stack. Here's a brief overview of each component and its role:

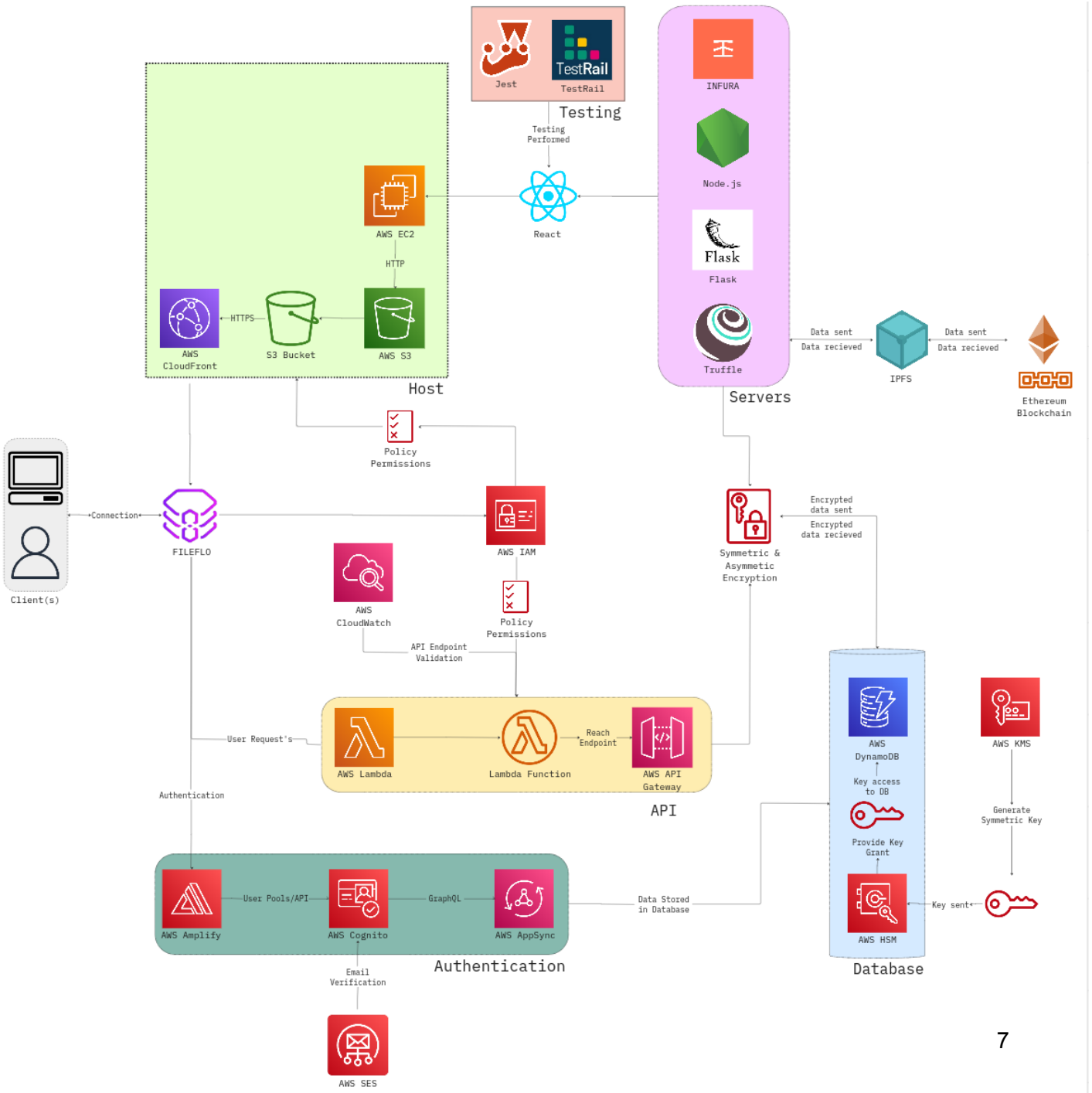
- **Client**: The end-user who interacts with the application.
- **Fileflo**: The core application that connects and interacts with various components to bring the application to life for the user.
- **Host**: Comprising AWS S3, AWS CloudFront, and AWS EC2, this component handles how Fileflo is hosted on the web, storing and deploying the React App for seamless user access.
- **Servers**: INFURA, Node.js, Flask, and Truffle power the necessary servers to create, fetch, and modify data for Fileflo's operation, supporting the React App and the database.
- **API**: This essential component links clients to our services, allowing them to create accounts and access our features. All data captured by the API is stored in DynamoDB for user access and monitoring.

- **Authentication**: This component manages account creation and enables clients to securely log in and use Fileflo's services.
- **Database**: As the final stage of the architecture, the database stores and maintains all user-generated data, including modifications and creations.
- **Testing**: This aspect ensures that the React App undergoes thorough testing to confirm all code components are functional and error-free before hosting.
- **Others**: AWS IAM serves as the heart of our architecture, providing necessary policies and grants for system communication. AWS CloudWatch detects incoming defects in the API, and AWS KMS supplies the database with symmetric keys to encrypt transmitted data.

Together, these components form a robust and efficient architecture for the Fileflo application.

[Link to full-scale image]

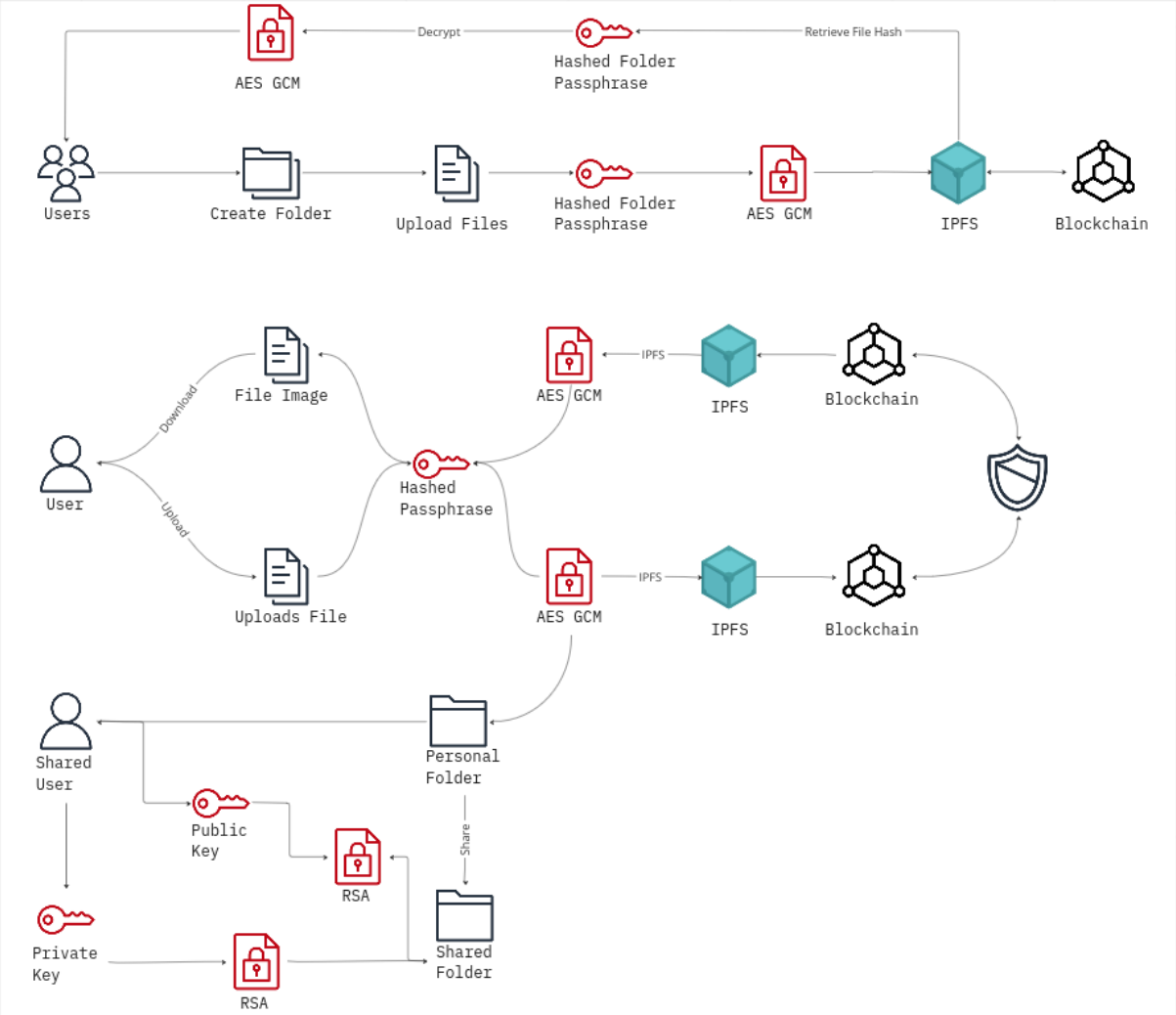
Fig 1.1 System Architecture



4.2 Data Flow

The Fileflo data flow diagram below details the process of uploading and downloading files, including the use of IPFS and the blockchain, as well as the encryption and decryption of files using AES/RSA. The diagram shows that when a user uploads a file, the file is first encrypted using the user's personal passphrase. The encrypted file is then uploaded to IPFS, which assigns a unique hash to the file. Next, the user's encrypted passphrase and the IPFS hash of the file are stored on the blockchain using a smart contract. The smart contract is used to manage the ownership and access control of the file. When a user wants to download a file, they must first authenticate themselves using their login credentials. Once authenticated, the user's private key is used to decrypt their personal passphrase, which is then used to decrypt the IPFS hash of the file stored on the blockchain. The decrypted IPFS hash is then used to retrieve the encrypted file from IPFS. The file is decrypted using the user's personal passphrase, and the user is able to access and download the file. This data flow diagram ensures that files are securely stored and accessed only by authorised users.

The Shared User portion of the diagram displays how a user can share a file or have a file shared to them and then be able to retrieve that file back. This process utilises RSA private and public key encryption to ensure only the shared user is able to retrieve that specific file from the owner. The Users section highlights how multiple users can effectively interact through collaborative file sharing. A folder is created among the users and a folder passphrase is made and hashed in order for the files to be encrypted and decrypted.



5. Implementation

5.1 Registration and Login

Registration

Registering a user for these three steps:

1. **Registration**
   First, a user needs to register an account by providing their name, username, email, and password. Each of these fields is validated to ensure they meet specific requirements, such as a unique username, matching passwords, a valid email, and a password that is at least 8 characters long with at least one number, one special character, and one uppercase character. This validation process, which can be done using regular expressions, helps create robust code that addresses security concerns and prevents unwanted bugs, ensuring a safe and secure experience for users.

   These are all validated through this Zod scheme:

```
// Validation schema
const schema = z
  .object({
    name: z.string().min(1, 'Name is required'),
    email: z
      .string()
      .email()
      .regex(
        new RegExp(/^[\w-.]+@([\w-]+\.)+[\w-]{2,4}$/),
        'Email address is not valid'
      ),
    password: z
      .string()
      .regex(
        new RegExp(
          /^(?=.*[a-z])(?=.*[A-Z])(?=.*\d)(?=.*[@$!%*?&])[A-Za-z\d@$!%*?&]{8,}$/
        ),
        'Password must contain: \n • Minimum of 8 characters \n • etc'
      ),
    repassword: z.string(),
  })
  .refine((data) => data.password === data.repassword, {
    message: "Passwords don't match",
    path: ['repassword'],
  });
```

2. **Confirm Verification**
   After a user has successfully registered, they are directed to the confirmation page to validate their account. This process involves Cognito sending an email with a unique code to the user's email address. The user must then enter the provided code, which is checked against the code sent by Cognito to confirm its validity. The email, sent

from "fileflotech@gmail.com," contains relevant information and guidance in case the user encounters any issues during this process.

3. **Request Passphrase**
   The final stage of user creation involves setting up a passphrase. After the user enters their chosen passphrase, it gets processed through the Flask backend to generate a pair of public and private keys. These keys are then securely stored in the database using AWS HSM for encryption, ensuring that the values remain protected. The private key is displayed to the user, allowing them to copy and save it in a secure location on their system.

Login

Users are prompted to enter their username and password to gain access to their account. If the provided credentials are valid, the user is successfully logged in, and a session is initiated. This session is created using JWT tokens, which store the user's information securely. If the Fileflo tab is closed or the user remains idle for an extended period, the token will reset, effectively logging the user out. This feature helps Fileflo ensure that user accounts remain protected from unauthorised access, even if someone gains control of the user's computer. To add, Amplify also keeps a storage of the users log in activity to ensure components such as File Upload, Sharing and Folder Creation are operational.

Below is a demonstration of

```
// login API call
try {
  const response = await loginUser(username, password);
  setUserSession(response.user, response.token);
  navigate('/Home');
} catch (error) {
  // Handle API errors
  if (
    error.response &&
    (error.response.status === 401 || error.response.status === 403)
  ) {
    setErrorMessage(error.response.data.message);
  } else {
    setErrorMessage(
      'Sorry....the backend server is down. Please try again later!!'
    );
  }
}

// Authenticate with AWS Amplify
try {
  await Auth.signIn(username, password);
} catch (error) {
  console.log('error signing in', error);
}
};
```

5.2 Navbar

The Navbar is a key component within the `App.js` file, as it appears on every single page. The following example illustrates the pages displayed in the Navbar, along with their respective links:

```
// Page names
const pages = ['Home', 'Upload', 'About'];
const profile = ['Profile', 'Add account', 'Logout'];
// Profile links
const profileLinks = {
    'Profile': '/profile',
    'Add account': '/register',
    'Logout': '/login',
};
```

Page Mapping

The most efficient approach to making navigation pages functional is by mapping them to corresponding links. The example below illustrates how this can be achieved:

```
{ /* Mapped Pages */}
{pages.map((page) => (
    <MenuItem key={page} onClick={handleCloseNavMenu}>
        <Typography textAlign="center">
            <NavLink
                className={(navData) => (navData.isActive ? "active-style" : 'none')}
                style={{
                    marginRight: "22px",
                    color: "white",
                    fontSize: "18px",
                    fontFamily: 'space-grotesk '
                }}
                exact="true"
                to={`/${page === "Upload" ? "folder/Personal" : page}`}
            >
                {page}
            </NavLink>
        </Typography>
    </MenuItem>
))}
```

Profile Mapping

The Profile button functions similarly to the pages mapping section, but instead of being a simple clickable component leading to a page, it serves as a dropdown menu. This menu

allows users to choose between logging out, creating an account, and viewing their profile page. Additionally, it checks if a username exists within the application, and if so, the user's username is displayed as the profile header.

```
    {username ? username : "Profile"}
</Button>
<Menu
    anchorEl={anchorEl}
    open={Boolean(anchorEl)}
    onClose={handleClose}
    MenuListProps={{
        'aria-labelledby': 'basic-button',
    }}
>
    {profile.map((page) => (
        <MenuItem sx={{fontSize: "22px"}} key={page} onClick={handleClose}>
            <NavLink
                className={(navData) => (navData.isActive ? 'active-style' : 'none')}
                style={{
                    textDecoration: 'none',
                    color: 'white',
                    fontSize: '18px',
                    '@media (max-width: 600px)': {
                    fontSize: '1.2rem',
                    },
                }}
                to={profileLinks[page]}
            >
                {page}
            </NavLink>
        </MenuItem>
    ))}
```

Private Routes

All navbar components serve as clickable links, but they are subject to a condition: they can be either public or private. Private routes help prevent unauthorised users from accessing pages such as Upload, Profile, and About. When a user logs in, their token is verified, and they gain access to public routes and all other routes. The function below demonstrates how a private route operates using a user's token:

```jsx
// Built-in imports
import React from 'react';
import { Navigate, Outlet } from 'react-router-dom';
// External imports
import { getToken } from '../service/AuthService';

// PrivateRoute component
const PrivateRoute = () => {
  return (
      getToken() ? <Outlet/>
        : <Navigate to="/login" />
  )
}

export default PrivateRoute
```

5.3 Blockchain Smart Contract

```
pragma solidity ^0.8.0;

contract FileMetadata {
    struct File {
        string fileName;
        string fileType;
        uint256 fileSize;
        uint256 uploadTime;
        string ipfsHash;
    }

    mapping (string => File) private files;

    function addMetadata(string memory fileName, string memory fileType, uint256 fileSize, string memory ipfsHash) public {
        uint256 uploadTime = block.timestamp;
        files[ipfsHash] = File(fileName, fileType, fileSize, uploadTime, ipfsHash);
    }

    function getMetadata(string memory ipfsHash) public view returns (string memory, string memory, uint256, uint256, string memory) {
        File memory file = files[ipfsHash];
        return (file.fileName, file.fileType, file.fileSize, file.uploadTime, file.ipfsHash);
    }
}
```

The FileMetadata smart contract seen above is a Solidity contract used to store metadata about files, including the filename, file type, size, upload time, and IPFS hash. The contract has a struct called File, which consists of the aforementioned file metadata properties. It also has a mapping that uses the IPFS hash as the key and the File struct as the value, allowing for easy access to the file metadata given an IPFS hash. The contract has two functions: addMetadata and getMetadata. The addMetadata function takes the file metadata properties as input and stores them in the File struct. The function also records the upload time as the block timestamp when the transaction is added to the blockchain. The getMetadata function takes an IPFS hash as input and returns the file metadata associated with it. The function retrieves the File struct associated with the IPFS hash from the mapping and returns its properties.

5.4 File Upload

The most important function for file upload is the addFile function, which encrypts the file, uploads it to IPFS, adds it to the database, and uploads the file metadata to the blockchain. The code snippet below shows how the addFile function uses the postEncryptedFile function to encrypt the uploaded file using the encryptedPassphrase. The encrypted data is then uploaded to IPFS using the ipfs.add() method.

```
const addFile = async (file) => {
   // Encrypt the file
   console.log('Encrypting the file...');
   const responseData = await postEncryptedFile(file, encryptedPassphrase);
   if (!responseData) {
     console.error('Failed to post encrypted file');
     return;
   }


   const encryptedData = responseData.data.hex_data;
   console.log('File encrypted successfully.');

   // Upload the encrypted file to IPFS
   console.log('Uploading the encrypted file to IPFS...');
   const result = await ipfs.add(encryptedData);
   console.log('File uploaded to IPFS successfully:', result);
```

The resulting IPFS hash is stored in the currentHash variable. The metadata of the uploaded file is then stored on the blockchain using the uploadMetadataToBlockchain function, which can be seen in the code snippet below. This function uses the contract instance to encode and sign a transaction, which adds the file metadata to the blockchain. The contract.methods is used to call the addMetadata function, which takes in the metadata information and returns an ABI-encoded transaction. This transaction is then signed by the uploader's private key, and the resulting signed transaction is sent to the blockchain using the web3.eth.sendSignedTransaction() method.

```
// Upload metadata to the blockchain function
const uploadMetadataToBlockchain = async (metadata) => {
  console.log('Uploading metadata to the blockchain...');
  const privateKey = process.env.REACT_APP_PRIVATE_KEY;
  const account = web3.eth.accounts.privateKeyToAccount(privateKey);

  const data = contract.methods
    .addMetadata(
      metadata.fileName,
      metadata.fileType,
      metadata.fileSize,
      metadata.ipfsHash
    )
    .encodeABI();

  const gas = await contract.methods
    .addMetadata(
      metadata.fileName,
      metadata.fileType,
      metadata.fileSize,
      metadata.ipfsHash
    )
    .estimateGas({ from: account.address });

  const gasPrice = web3.utils.toWei('2', 'gwei');
  const tx = {
    from: account.address,
    to: contractAddress,
    gas,
    gasPrice,
    data,
  };

  console.log('Signing the transaction...');
  const signedTx = await account.signTransaction(tx);
  console.log('Sending the signed transaction...');
  return web3.eth.sendSignedTransaction(signedTx.rawTransaction);
};
```

5.5 File Sharing

The code snippet below shows part of the formSubmit function, which is responsible for handling the form submission that occurs when a user wants to share a file or folder with another user. The function begins by checking the validity of the user input. It checks that an owner has been specified to share the file/folder with, and that a passphrase has been entered. If any of these checks fail, an error message is displayed to the user. Next, the function checks if the folder path is a Personal or Shared folder. If it is not, it checks if the entered folder passphrase is correct using the folderPassphraseCheck function. If the passphrase is correct, the folder is added to the user being added, and the user is added to the file using the addUser function. If the folder path is a Personal or Shared folder, the function retrieves the public key of the user being added using the retrieveUserPublicKeyAPI function. It then hashes the user's personal passphrase and encrypts it using the user's

public key with the encryptPassphrase function. The resulting encrypted passphrase is then added to the user being added, and the user is then added to the file using the addUser function.

```
// Check if the folder path is a Personal or Shared folder
if (folderPath !== "Personal" && folderPath !== "Shared") {
  const response = await folderPassphraseCheck(user.username, folderPath, folderPassphrase)
  if (response.error) {
      setErrorMessage(response.error);
      return;
  }
  // Add folder with passphrase to user being added
  await submitFolderPassphrase(user.username, owner, folderPath, folderPassphrase, null, setErrorMessage);
  // Add user to the file
  await addUser();
  setSuccessMessage('User Successfully Added')
} else {

  if (owner.trim() === '') {
    setErrorMessage('Please input an owner to share with');
    return;
  }
  setErrorMessage(null);
  if (passphrase.trim() === '') {
    setErrorMessage('Passphrase should not be empty');
    return;
  }
  const error = await enterPassphrase(passphrase);
  if (error) {
      setErrorMessage(error);
      return;
  }
```

5.6 File Download

The automaticFileDownload function seen below downloads a file by first confirming the user's passphrase and then fetching the file from IPFS using the supplied hash. It determines the passphrase to use for decryption based on whether the user is the owner of the file, the passphrase for the owner, or the passphrase for the folder the file is in. The file is then decrypted using Flask and the decrypted bytes are converted to a file and initiated for download. Session storage and state are updated accordingly, and if an error occurs, the function sets the error message.

```
const automaticFileDownload = async (
  ipfsHash,
  newFilename,
  fileType,
  updatedFolderPassphrase
) => {
  try {
    // Confirm passphrase
    const response = await confirmPassphrase(user.username, 'test');

    // Check if the response is not null
    if (!response) {
      console.error('Failed to confirm passphrase');
      return;
    }

    // Fetch file from IPFS
    const IPFSresponse = await fetch(ipfsFileURL + ipfsHash);
    const IPFSdata = await IPFSresponse.text();
    formData.append('hex_data', IPFSdata);

    // Determine the final passphrase to use for decryption
    let finalPassphrase;
    if (isOwner) {
      finalPassphrase = user.passphrase[0];
    } else {
      finalPassphrase = ownerPassphrase;
    }
    if (folderPath !== 'Personal' && folderPath !== 'Shared') {
      finalPassphrase = updatedFolderPassphrase;
    }

    // Decrypt the file
    const flaskResponse = await decryptFile(finalPassphrase, fileType);

    // Convert the decrypted bytes to a file and initiate the download
    const blob = new Blob([flaskResponse.data], { type: `\${fileType}` });
    const link = document.createElement('a');
    link.href = URL.createObjectURL(blob);
```

The getFileIPFSHashFromBlockchain function uses the Web3 library to interact with a smart contract on the Ethereum blockchain. The function retrieves the IPFS hash, file name, and file type associated with a selected file by calling the smart contract method getMetadata with the selected file hash as a parameter. The retrieved IPFS hash, new filename, and file type are returned by the function. If an error occurs, an error message is logged and the function returns null. The function also logs the contract ABI and address, and the selected file hash for debugging purposes.

```
// Retrieve the IPFS hash from the smart contract
const getFileIPFSHashFromBlockchain = async () => {
  try {
    const web3 = new Web3(
      new Web3.providers.HttpProvider(
        'https://sepolia.infura.io/v3/7720f63fce8443e5b85e0acc7356c0cf'
      )
    );

    // Get the contract ABI and address
    const contractABI = FileMetadata.abi;
    const networkId = '11155111';
    const contractAddress = FileMetadata.networks[networkId].address;
    const contractInstance = new web3.eth.Contract(
      contractABI,
      contractAddress
    );

    // Log the contract ABI and address
    console.log('Contract ABI:', contractABI);
    console.log('Contract Address:', contractAddress);

    console.log('selectedFileHash:', selectedFileHash);
    const accounts = await web3.eth.getAccounts();

    // Get the IPFS hash from the smart contract
    const ipfsHash = await contractInstance.methods
      .getMetadata(selectedFileHash)
      .call({ from: accounts[0], gas: 3000000 });
    console.log(ipfsHash);

    // Return the correct IPFS hash, new filename, and file type
    return {
      ipfsHash: ipfsHash[4],
      newFilename: ipfsHash[0],
      fileType: ipfsHash[1],
    };
```

5.7 Folder Creation

Users can create folders through the upload page of the application. The process involves taking a folder name as input and then storing the folder name under the folder attribute in the DynamoDB File table. This is achieved using GraphQL mutations.

Example of Folder Creation:

```
// GraphQL Mutation
const createFolder = async () => {
  try {
    const folderData = {
      folder: folder.replace(/\s+/g, '-'),
    };
    await API.graphql(graphqlOperation(createFile, { input: folderData }));

    window.location.reload();
  } catch (err) {
    console.log('error creating Folder:', err);
  }
};

// GraphQL Query
const fetchFolders = async () => {
  try {
    const result = await API.graphql(graphqlOperation(listFiles));
    const folders = result.data.listFiles.items.map((item) => item.folder);
    setFolders(folders);
  } catch (err) {
    console.log('error fetching folders:', err);
  }
};
```

5.8 AES Algorithm

File Encryption

In Fileflo, every uploaded file is processed through a Flask endpoint to be encrypted into hexadecimal data using AES symmetric encryption. The endpoint function accepts the file and a passphrase as inputs, combining these values to generate hexadecimal data corresponding to the uploaded file. The `encrypt` function performs file encryption by generating a random salt and deriving a key from the provided password and salt using the scrypt key derivation function. It then uses the Advanced Encryption Standard (AES) in Galois/Counter Mode (GCM) mode to encrypt the file data in chunks. The encrypted chunks are represented as hexadecimal data and concatenated to create the final encrypted file. This process ensures that the file is securely encrypted and can only be accessed with the correct password.

```python
# Encrypt a file using a password
@app.route('/encrypt', methods=['POST'])
def encrypt():
    """
    Encrypt a file using a password

    :return: The encrypted file as a hex string
    """
    # Get the password from the POST request
    password = request.form['passphrase']

    # Get the file data from the POST request
    file_data = request.files['file'].read()

    salt = get_random_bytes(32)  # Generate salt
    key = scrypt(password, salt, key_len=32, N=2**17, r=8, p=1)  # Generate a key using the password and salt

    encrypted_data = b''

    # Create a cipher object to encrypt data
    cipher = AES.new(key, AES.MODE_GCM)  # Create a cipher object to encrypt data
    encrypted_data += salt  # Add the salt to the encrypted data
    encrypted_data += cipher.nonce  # Add the nonce to the encrypted data

    # Encrypt the data
    data_len = len(file_data)
    for i in range(0, data_len, BUFFER_SIZE):
        data_chunk = file_data[i:i+BUFFER_SIZE]
        encrypted_chunk = cipher.encrypt(data_chunk)  # Encrypt the data we read
        encrypted_data += encrypted_chunk  # Add the encrypted chunk to the encrypted data

    # Signal to the cipher that we are done and get the tag
    tag = cipher.digest()

    encrypted_data += tag  # Add the tag to the encrypted data

    # Convert the encrypted data to hex and return the response
    hex_data = binascii.hexlify(encrypted_data)
    return jsonify({'data': {'hex_data': hex_data.decode()}})
```

File Decryption

The file decryption is essentially the reverse of File Encryption. It takes in a passphrase, the hexadecimal data and file type parameters. It then converts the hex-encoded data to bytes and extracts the salt, nonce, and tag from the encrypted data. Then, it generates a key using the scrypt function with the password and salt. Using the key and nonce, it creates an AES cipher object in GCM mode to decrypt the data. After decryption, the function verifies the tag to ensure successful decryption. Finally, the decrypted data is returned as a file with an appropriate file extension, allowing the user to download it. This is shown below:

```python
# Decrypt a file using a password
@app.route('/decrypt', methods=["POST"])
def decrypt():
    """
    Decrypt a file using a password

    :return: The decrypted file as a hex string
    """
    password = request.form.get('passphrase') # Get the password from the POST request
    hex_data = request.form.get('hex_data')
    file_type = request.form.get('file_type')

    # Convert the hex data to bytes
    encrypted_data = binascii.unhexlify(hex_data)

    salt = encrypted_data[:32]  # Get the salt from the encrypted data
    key = scrypt(password, salt, key_len=32, N=2**17, r=8, p=1)  # Generate a key using the password and salt

    nonce = encrypted_data[32:48]  # Get the nonce from the encrypted data
    cipher = AES.new(key, AES.MODE_GCM, nonce=nonce)  # Create a cipher object to decrypt data

    # Decrypt the data
    data = encrypted_data[48:-16]  # Get the encrypted data
    decrypted_data = cipher.decrypt(data)  # Decrypt the data

    # Verify the tag for decryption verification
    tag = encrypted_data[-16:]  # Get the tag from the encrypted data
    try:
        cipher.verify(tag)  # Verify the tag
        print("Tag verification successful")
    except ValueError:
        print("Tag verification failed")

    # Return the decrypted data
    file_extension = mimetypes.guess_extension(file_type)
    decrypted_data_file = io.BytesIO(decrypted_data)
    download_name = f'decrypted_file{file_extension}'
    return send_file(decrypted_data_file, mimetype=file_type, as_attachment=True, download_name=download_name)

```

5.9 RSA Algorithm

Generate RSA Public Key and Private Key

RSA was used to create safe and secure Public and Private keys to then allow users to share and encrypt/decrypt files safely. The RSA_generate_pairs() function simply generates a pair of Public and Private keys by generating a 2048-bit private key with a public exponent of 65537. A public key is then derived from the private key. Both the private and public keys are serialised to PEM format using the `private_bytes()` and `public_bytes()` methods, respectively. Finally, the function returns the private and public keys in a JSON response, with the keys decoded from bytes to UTF-8 strings. This is shown below:

```python
# Generate a pair of RSA keys
@app.route('/generate_keys', methods=['GET'])
def RSA_generate_pairs():
    """
    Generate a pair of RSA keys

    :return: The private and public keys as a JSON response
    """
    # Generate a private key
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048,
        backend=default_backend()
    )

    # Generate a public key from the private key
    public_key = private_key.public_key()

    # Serialize the private key to PEM format
    pem_private_key = private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.NoEncryption()
    )

    # Serialize the public key to PEM format
    pem_public_key = public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )

    # Return the keys as a JSON response
    return jsonify({
        'private_key': pem_private_key.decode('utf-8'),
        'public_key': pem_public_key.decode('utf-8')
    })
```

PEM Checker

The code snippet provided demonstrates how PEM validation works by utilising regex to compile the string and verify that all required string values are present with the correct indentation and syntax. This process ensures that the keys are formatted correctly and ready for use within the Flask application.

```python
# Check if a string is a valid PEM format
def is_pem(pem_string):
    """
    Check if a string is a valid PEM format

    :param pem_string: The string to check
    """
    pattern = re.compile(
        r'^-----BEGIN [A-Z ]+-----\n'
        r'[a-zA-Z0-9/+=\n]+'
        r'^-----END [A-Z ]+-----\n?$',
        re.MULTILINE | re.ASCII
    )
    return bool(pattern.match(pem_string))
```

PEM Format

This code snippet shown below, on the other hand, performs the inverse operation by taking in a PEM string and formatting it correctly. After formatting the PEM string, it runs the formatted string through the PEM validation function to ensure it meets the necessary PEM format requirements. This function is particularly useful when users input their private keys on the front-end, as it ensures that these private keys are formatted correctly before processing them further in the application.

```python
# Format a private key to PEM format
def format_private_key_pem(private_key_str):
    """
    Format a private key to PEM format

    :param private_key_str: The private key to format
    """
    header = "-----BEGIN PRIVATE KEY-----\n"
    footer = "\n-----END PRIVATE KEY-----"
    key_body = private_key_str[len(header):-len(footer)].replace(" ", "")  # Remove extra spaces
    formatted_key_body = ''.join([key_body[i:i + 64] + '\n' for i in range(0, len(key_body), 64)]).rstrip()
    return header + formatted_key_body + footer
```

Encrypt/Decrypt Passphrase

RSA also had the challenge of encrypting and decrypting the hashed passphrases. The `encrypt passphrase` function takes a public key and a hashed passphrase as input, loads the public key in PEM format, and encrypts the hashed passphrase using the OAEP padding scheme with SHA-256 hashing. It then returns the encrypted passphrase as a base64-encoded string. The `decrypt passphrase` function on the other hand takes a private key and an encrypted passphrase as input, validates and formats the private key if necessary, and then loads it in PEM format as mentioned above. It decrypts the encrypted passphrase using the same OAEP padding scheme with SHA-256 hashing and returns the decrypted passphrase as a plain string.

Example of the encryption of hashed passphrase:

```python
# Encrypt a passphrase using a public key
@app.route('/encrypt_passphrase', methods=['POST'])
def encrypt_passphrase():
    """
    Encrypt a passphrase using a public key

    :return: The encrypted passphrase as a base64-encoded string
    """
    # Get the public key and hashed passphrase from the request
    public_key_pem = request.json['public_key']
    hashed_passphrase = request.json['hashed_passphrase']

    # Load the public key from its PEM format
    public_key = serialization.load_pem_public_key(
        public_key_pem.encode(),
        backend=default_backend()
    )

    # Encrypt the hashed passphrase using the public key
    encrypted_passphrase = public_key.encrypt(
        hashed_passphrase.encode(),
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

    # Return the encrypted passphrase as a base64-encoded string
    return jsonify({"encrypted_passphrase": base64.b64encode(encrypted_passphrase).decode()})
```

5.10 AWS

GraphQL / AppSync

The integration of GraphQL in the project was facilitated using AWS Amplify, which helped create a GraphQL API. This resulted in the generation of multiple files containing essential functional components, such as mutations and subscriptions, enabling the creation and modification of query requests. These files are then utilised within the

`src/pages/Upload/Upload.js` file, where data retrieved from the database using GraphQL is transmitted.

Here's an example of how two subscriptions (update and delete) are invoked. First, we import the `graphqlOperations` and then set the stored variables to the appropriate state for further use in the code:

```
const updateSub = API.graphql(
  graphqlOperation(onUpdateFile, subscriptionFilter)
).subscribe({
  next: ({ value }) => {
    setFiles((Files) => {
      const toUpdateIndex = Files.findIndex(
        (item) => item.id === value.data.onUpdateFile.id
      );
      // If the File doesn't exist, treat it like an "add"
      if (toUpdateIndex === -1) {
        return [...Files, value.data.onUpdateFile];
      }
      return [
        ...Files.slice(0, toUpdateIndex),
        value.data.onUpdateFile,
        ...Files.slice(toUpdateIndex + 1),
      ];
    });
  },
});

const deleteSub = API.graphql(graphqlOperation(onDeleteFile)).subscribe({
  next: ({ value }) => {
    setFiles((Files) => {
      const toDeleteIndex = Files.findIndex(
        (item) => item.id === value.data.onDeleteFile.id
      );
      return [
        ...Files.slice(0, toDeleteIndex),
        ...Files.slice(toDeleteIndex + 1),
      ];
    });
  },
});
```

API Requests

A significant aspect of the AWS implementation involved creating endpoints for variables to reach their corresponding endpoints. To accomplish this, we established `src/utils/API` and `src/utils/KMS` directories to house all the POST and GET methods, along with the necessary components for executing each request. Consequently, in each file or function, we invoked the endpoint function and passed the desired variables.

Filetree

```
src
└── utils
        └── API
                └── retrievePrivateKey.js
```

```
        └── retrievePublicKey.js
        └── login.js
        └── …
├── AWS
        ├── KMS.js
```

Here is a demonstration of what one of these files may look like:

```
// External Dependencies
import axios from "axios";

// Internal Dependencies
import { requestConfig } from "./requestConfig";

// retrievePrivateKeyAPI function
export const retrievePrivateKeyAPI = async (username) => {
    // request body
    const requestBody = {
      username: username,
    };
    const retrievePrivateKeyAPIURL =
      'https://s41eemifi8.execute-api.eu-west-1.amazonaws.com/prod/retrieveuserprivatekey';

    try {
      const response = await axios.post(
        retrievePrivateKeyAPIURL,
        requestBody,
        requestConfig
      );
      return response.data.privateKey;
    } catch (err) {
      console.error('Error retrieving user private key:', err);
      return null;
    }
};
```

Lambda

With API files in mind, another section to AWS that had to be provided was corresponding lambda functions within AWS that acted as the middle layer between the database and the backend. These lambda functions were invoked by the API requests and they were stored in AWS and updated to be in src/backend/lambda.

File Tree as an example

```
src
└── backend
        └── lambda
                └── services
                        └── confirmPass.js
                        └── login.js
                        └── …
                └── utils
                └── index.js
```

6. Problems Solved

6.1 Moving from Local to Hosted Blockchain

We initially used a local Ganache Ethereum blockchain, which we had to run on localhost:7545 every time we wanted to test anything blockchain-related, such as file upload and download. Unfortunately, during the last step of our blockchain download setup, we kept encountering the following error message every time we tried to download a file: '*Error: Returned values aren't valid, did it run Out of Gas?*'. We spent several days trying to resolve this problem and tried several solutions, such as updating the smart contract's ABI and running a different version of Ganache, but none of them seemed to work. The error message suggested that the issue might be related to running out of gas, but increasing the gas limit did not solve the problem either. We also checked to ensure that we were requesting data from the correct block number and that the node was fully synced, but still encountered the same error.

We were left with one possible solution, which was to migrate from our local blockchain to a hosted blockchain. We had always intended to host our blockchain so that it would not require a localhost instance in order to function, however the error forced us into doing this earlier than anticipated. After researching alternatives to Ganache, we learned that we would need to use a different Ethereum node provider, which is where we encountered Infura's Ethereum testnet service. We were already using Infura's IPFS API, so we configured our existing Infura account to use the Ethereum Sepolia testnet endpoint. After editing our smart contracts and Truffle configuration files, we were successfully able to retrieve data from the smart contract without encountering the same error. The fact that simply migrating to Infura's Sepolia blockchain fixed our issue proved that the error was not actually caused by gas fees, but rather by a Ganache incompatibility. While we were disappointed that we were not able to resolve the issue with our local Ganache blockchain, we were ultimately relieved to have found a solution that allowed us to continue developing our application, whilst also implementing a blockchain that is always available to users.

6.2 User Sharing Issues

The initial process of sharing files among users was straightforward, as it simply involved utilising a GraphQL schema to store user data and then transferring that information between users when they shared a common attribute. However, as we introduced various file-sharing methods, such as personal files, folder files, different passphrases, and the use of public and private keys, ensuring seamless file sharing and recipient access without any issues became a complex task.

The primary challenge stemmed from the numerous conditions associated with each file. These conditions included factors such as the file's location in a specific folder, whether the user was the owner or not, and the sharing of folder passphrases among multiple users when folders were nested within other folders.

To overcome this issue, we employed a meticulous approach, integrating conditional statements for each file-sharing scenario to verify the environment and apply the appropriate

user-sharing method based on the given conditions. This process entailed monitoring various variables and endpoints, as well as ensuring accurate data transfer between the database and the backend.

6.3 Incorrect Encryption Method

Initially, we embarked on a journey to explore AES encryption and identify the most effective method to safeguard data. This involved a series of trial and error experiments with various functions, methods, and components.

After determining the most suitable encryption technique, we integrated it into the backend. However, we encountered several challenges related to the encryption function, which incorrectly encrypted data. The output hexadecimal data string length was incorrect, rendering decryption impossible. Sometimes, the function couldn't read the file to perform decryption, and other times, the decryption function failed due to passphrase issues or produced incorrect file types and data.

Overcoming these obstacles required extensive research, particularly on stackexchange.com, and multiple trial and error attempts. Eventually, we successfully developed an encryption and decryption function that leverages AES in GCM mode to securely encrypt and decrypt data without any issues.

6.4 Retrieving AWS Passphrases

In the early stages of the project, retrieving items from the DynamoDB was not a concern as the system was not yet scaled. However, as we introduced public and private keys and various methods of supplying passphrases to files for enhanced encryption security, we faced several challenges. These included properly storing passphrases, avoiding mix-ups, and retrieving the correct passphrase for the intended action. To address these issues, we had to add numerous conditions to each upload, share, and download action. These conditions checked if a user was the owner, if they were in the shared folder, personal folder, or a custom folder, and if a file or folder was shared with the user.

To resolve these complexities, we implemented Lambda functions that could retrieve the correct passphrase from the appropriate location with the right parameters and definitions. Throughout the process, fixing one component often led to another component breaking, requiring further adjustments. Overall, this iterative process was akin to a loop, where each action had to be conditioned multiple times to handle various passphrases, keys, and requirements.

7. Future Work

The future of Fileflo is promising, as there is real potential for our application to evolve and expand in various ways. One possibility for the future of Fileflo is to integrate more blockchain technologies, such as new smart contracts, to increase the efficiency and automation of the platform. By implementing smart contracts, Fileflo could enable more complex sharing arrangements, such as conditional access to files, based on predefined

rules or criteria. This would allow for even more secure and private sharing of files, while also reducing the need for intermediaries in the sharing process.

Another potential future feature for Fileflo could be an automatic folder syncing capability. This feature would allow users to create a shared folder and automatically sync all files uploaded to that folder with Fileflo. This would save users time and effort by eliminating the need to manually upload each file to the platform. To implement this feature, we would need to develop a file-watching service that constantly monitors the shared folder for new files. When a new file is detected, the service would automatically upload it to Fileflo and add it to the appropriate folder. The automatic folder syncing feature could be a significant improvement to the user experience on Fileflo, particularly for users who frequently collaborate with others on shared files.

The addition of a chat feature to Fileflo could greatly enhance the platform's collaborative capabilities and attract a wider user base. By enabling users to communicate and exchange ideas in real-time within the platform, the chat feature would allow for more seamless collaboration between individuals and groups. This could be particularly useful for businesses and organisations that require a secure and private platform for sharing sensitive information and collaborating on projects. Furthermore, the chat feature could also provide a forum for users to discuss and provide us with feedback on uploaded files, thereby enhancing the platform's overall functionality and user experience. This could be achieved by allowing users to leave comments and suggestions on files and folders, as well as the ability to discuss these files with other users in real-time. Such features would enable us to better meet the needs of our users and also help to differentiate Fileflo from other file-sharing platforms on the market.

A potential monetisation feature could be the introduction of a decentralised marketplace on Fileflo, where users could buy and sell files using Fileflo tokens. The marketplace could be designed to include a variety of files, ranging from multimedia content to software applications and other digital assets. For buyers, the marketplace would offer a diverse range of files that could be purchased securely and easily using Fileflo tokens. By leveraging the decentralised nature of the platform, buyers could also have confidence in the authenticity of the files they purchase, as they would be able to trace the provenance and history of each file on the blockchain. For sellers, the marketplace would provide a new revenue stream for their files, allowing them to monetise their content in a way that is secure, transparent, and fair. Additionally, the marketplace could be designed to include a feedback and rating system to ensure that high-quality files are rewarded and low-quality files are weeded out, thus creating a higher quality selection of files for buyers to choose from.

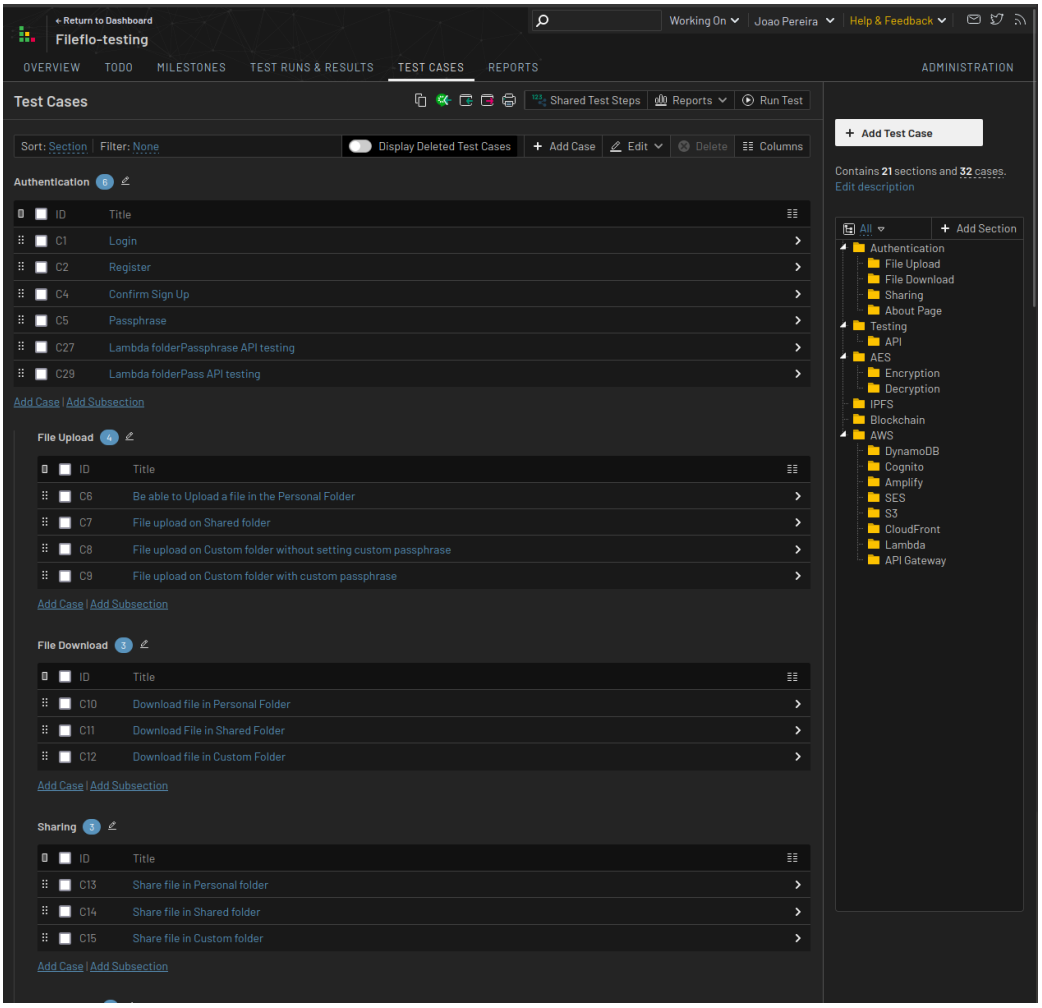
8. Testing

8.1 Manual Testing

Manual testing was an important part of our software development process. We utilised TestRail, a test management tool that enabled us to create a comprehensive suite of manual test cases. The test cases were designed to ensure that all components within the

application were working as expected and meeting the user requirements. Each time a branch was pushed to Git, we would run the corresponding test cases, marking each test as either a pass or fail depending on the outcome. This allowed us to quickly identify any issues that arose during development and address them promptly, ensuring that the application remained stable and functional. We performed a series of predefined tests that focused on the functionality and usability of the application. This included testing individual components, user interfaces, and overall system functionality.

We focused on testing a range of components, including the smart contract, user authentication, and file storage. We also tested the user interface to ensure that it was easy to use and that users could navigate the application with ease. Our manual testing process helped us to identify and fix bugs, improve the application's overall stability, and ensure that it met the user's expectations. Our TestRail can be seen below:



8.2 Ad Hoc Testing

Our ad hoc testing approach for Fileflo was a continuous process that we performed throughout the duration of the project. For example, in order to validate the application's upload feature, we used boundary value analysis techniques to test the application's ability

to handle different types of files of various sizes and formats. This involved uploading files of varying sizes, including large files, to determine if there were any limitations or restrictions on the size of files that could be uploaded. We also tested the application's ability to handle different file formats, such as text files, images, and PDFs, to ensure that users could upload files in any format they needed. We used exploratory testing techniques to upload different files at different times and in different orders to ensure that the application could handle these scenarios.

To test the application's file-sharing feature, we employed error guessing techniques to identify any defects in the feature. We intentionally attempted to share files with incorrect credentials or shared files with the wrong users to see how the application would handle these situations. Additionally, we tested the application's ability to store metadata on the blockchain, which included information such as the file owner, file type, and file location on IPFS. We also tested the application's ability to retrieve this metadata from the blockchain when necessary. We used exploratory testing techniques to upload different files and metadata to the network to ensure that the application could handle these scenarios.

8.3 User Evaluation

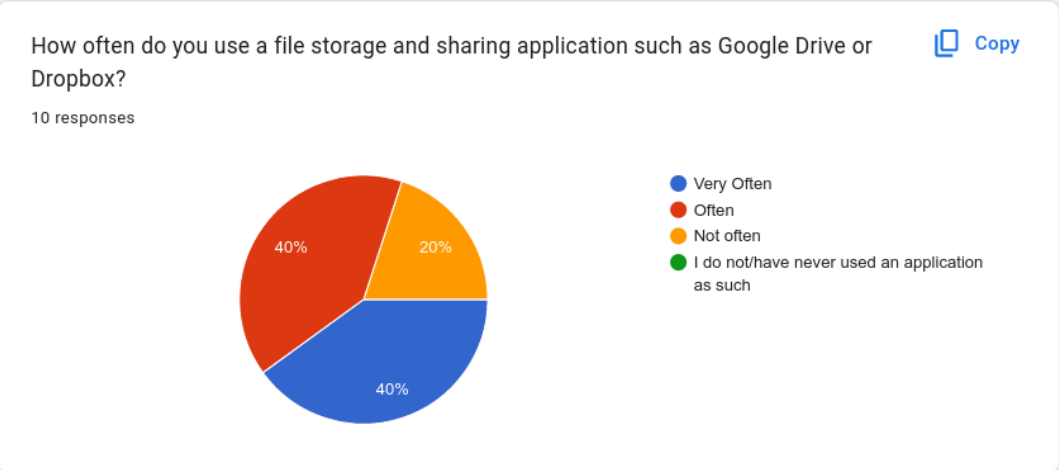
User evaluation played a crucial role in our testing process, offering invaluable insights into the user experience and allowing us to assess the application's performance from a fresh perspective. As the creators and developers of Fileflo, we have a comprehensive understanding of its components, concepts, and overall navigation. Nonetheless, our extensive familiarity with the application can be both a benefit and a drawback; while the application may be easy for us to use, it doesn't guarantee a similar experience for other users. To address this challenge, we conducted two forms of user testing, aiming to gauge the application's stability and comprehensibility for a diverse range of users. Our goal was to ensure a satisfying experience with minimal issues while facilitating smooth navigation throughout the application.

Incorporating user feedback and other evaluation techniques, we were able to identify potential usability issues and refine our application accordingly. This comprehensive evaluation process ensures Fileflo delivers a seamless, user-friendly experience that meets the needs of a diverse user base.

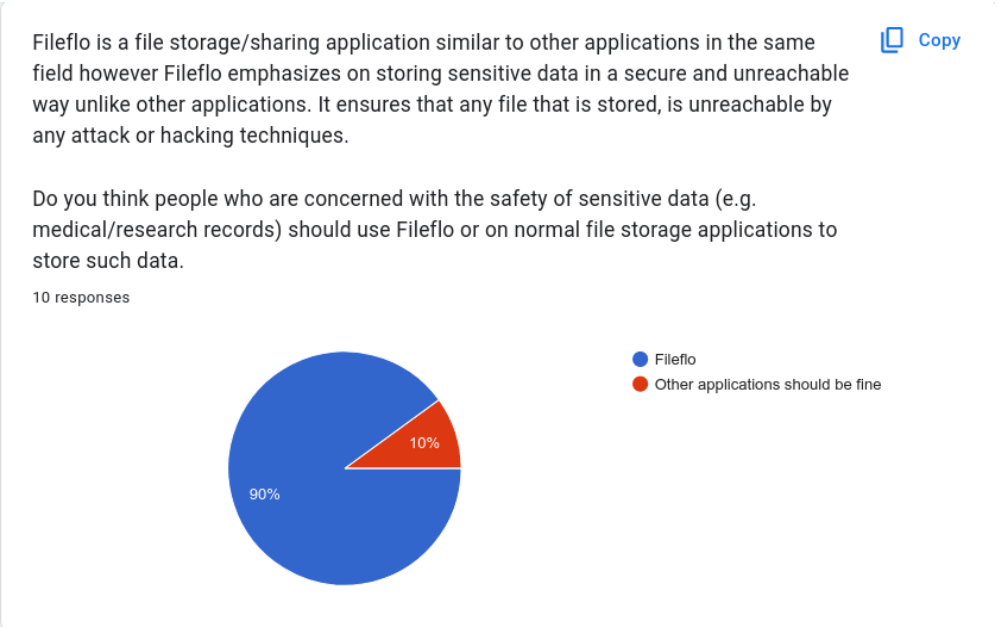
8.3.1 User Forms

Our initial approach to user testing involved sending a Google Form to a diverse group of individuals, featuring targeted questions related to Fileflo's concept, usability of components, and overall design and UI implementation. This method allowed us to gather insights about the application's direction and identify any necessary adjustments to better align with potential customer and user needs.

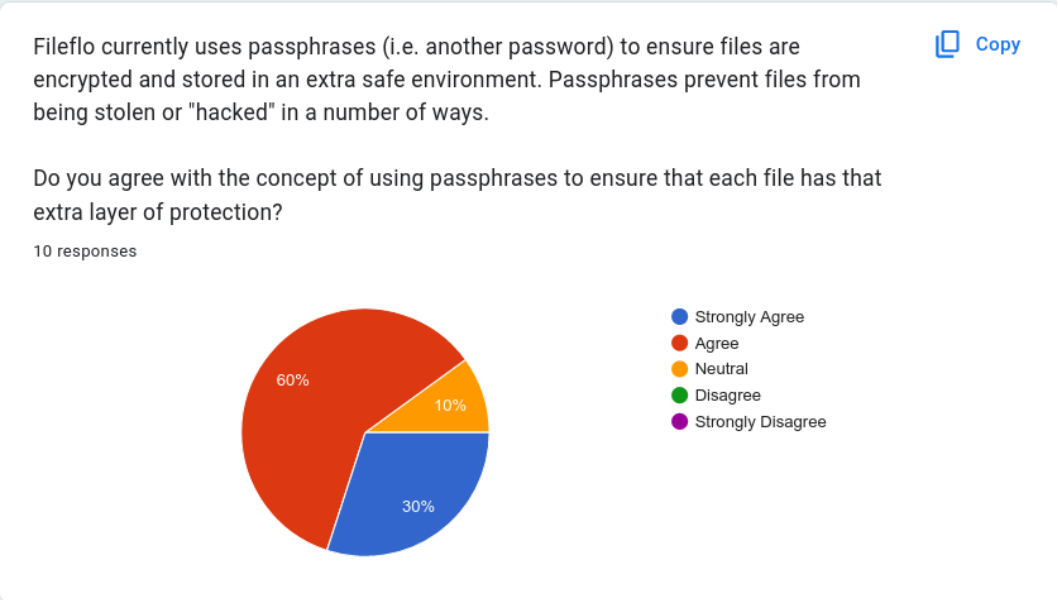
1. Our first question aimed to determine the number of users who actively utilise file storage and sharing applications. This insight helped us assess whether subsequent questions might be biased or influenced by users' unfamiliarity with such applications.



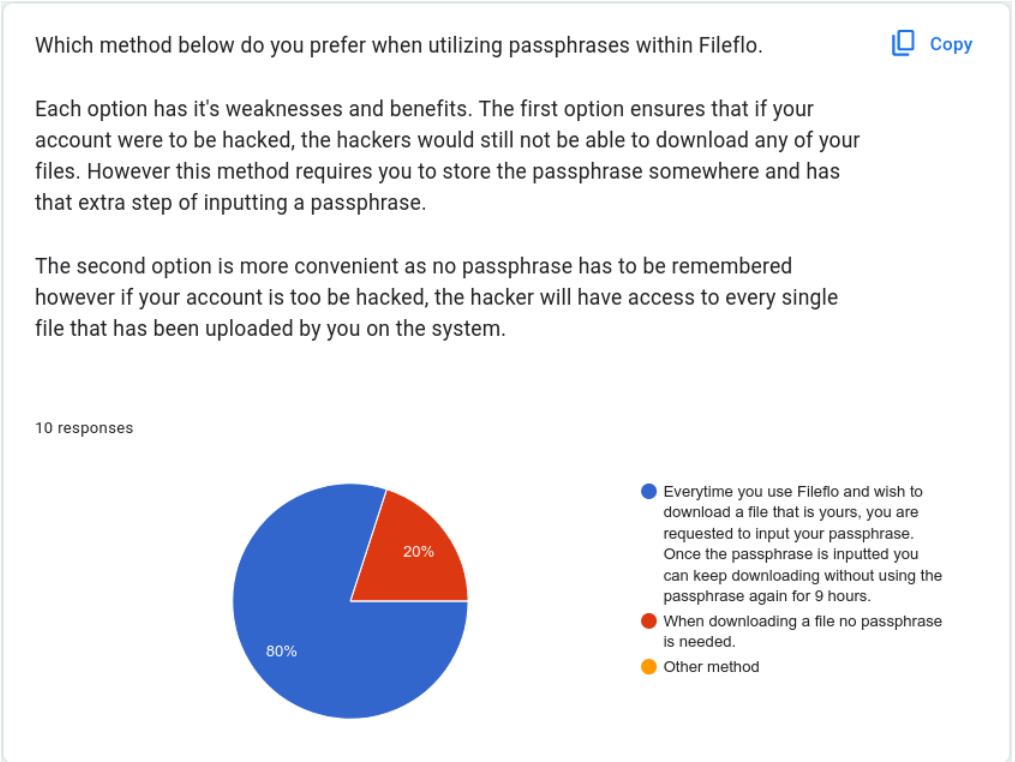
2. Our second question sought to gauge whether individuals believed that our application's approach to handling sensitive data was superior when compared to other applications and software in the same industry. The results show that 90% of respondents prefer Fileflo over alternative applications, indicating a positive trajectory for our application's direction.



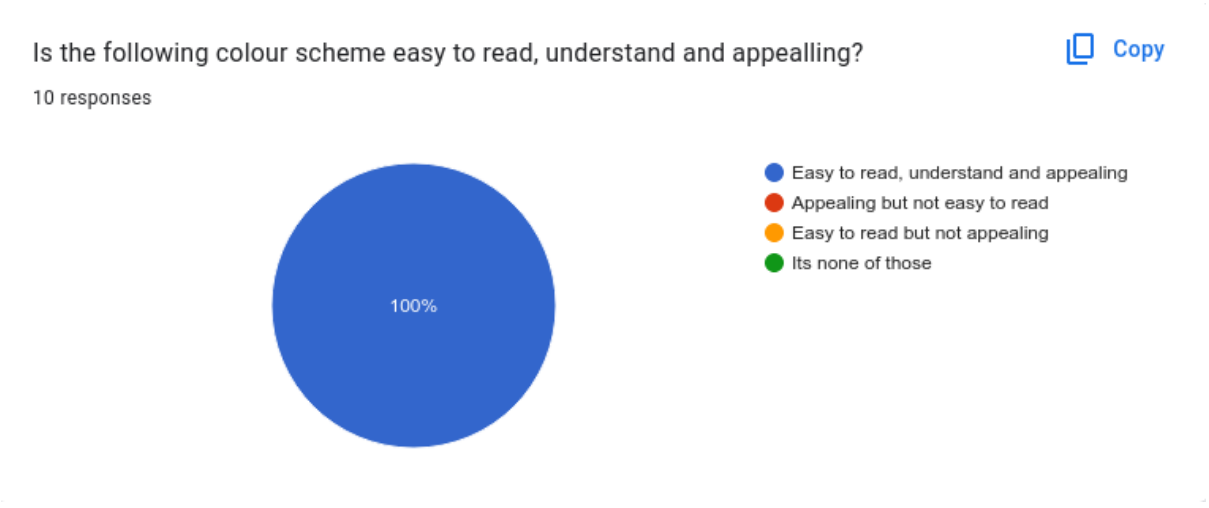
3. The survey explored respondent's perspectives on the adoption of passphrases, with a striking 90% expressing their approval. This strong endorsement guided us to recognise passphrases as a crucial component in our system.



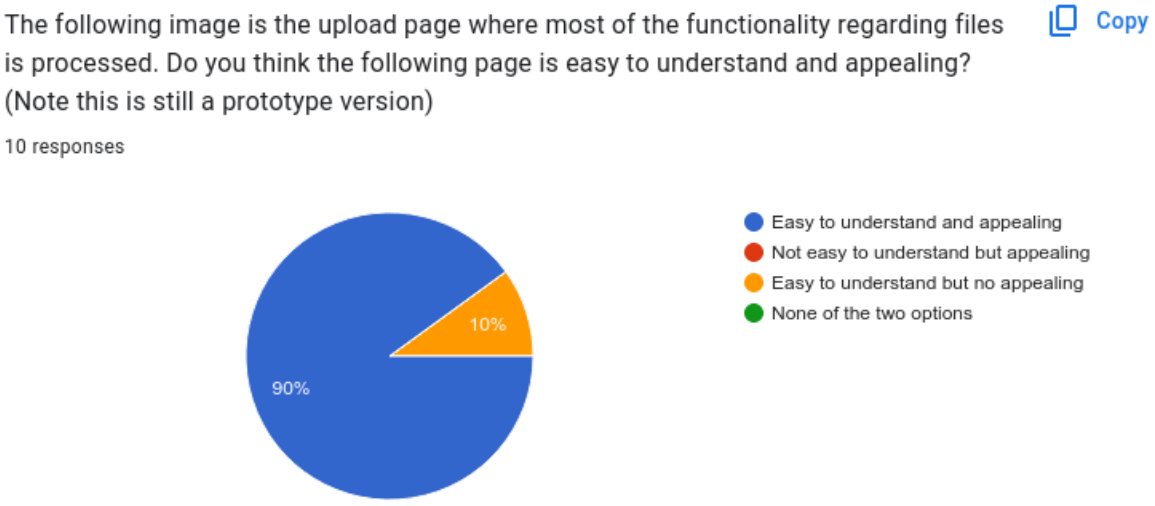
4. The central question we aimed to explore was the user perception of passphrases as both a user-friendly and effective method for ensuring maximum security of their files, as opposed to a less secure alternative without the use of passphrases. Passphrases are a crucial component of Fileflo, acting as the ultimate barrier between the user and their file, and are heavily emphasised within the application. Our team and supervisor had previously debated the necessity and user responsibility of relying on passphrases. With 90% of respondents in favour of using passphrases, the results clearly indicated that passphrases should remain an integral feature of Fileflo.



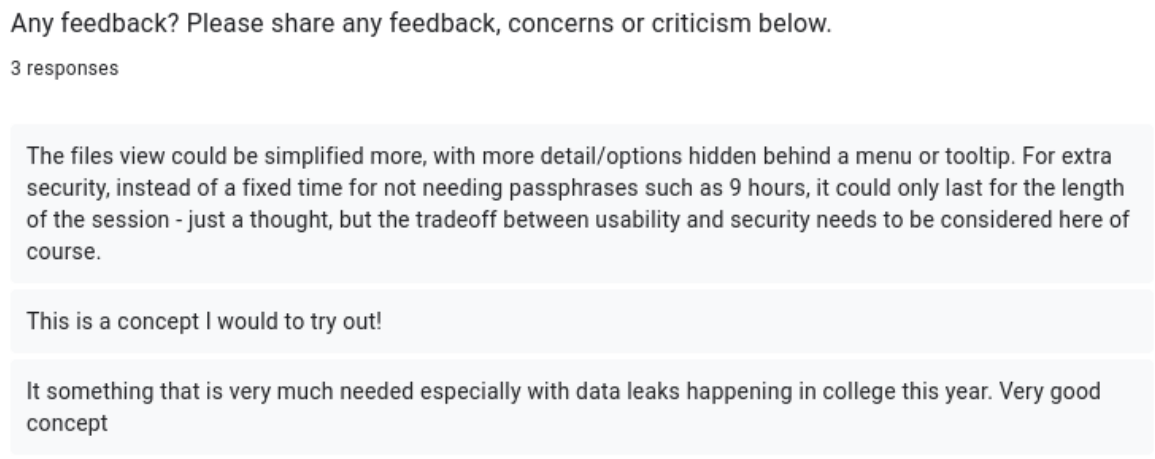
5. Our next question was related to the user interface, particularly the colour scheme. We wanted to have a unique, modern look to our application, whilst also prioritising the user experience. The responses showed a unanimous opinion that the application's colour scheme did in fact achieve this.



6. The next question concerned the upload page of our application, which is undoubtedly the most important page and the one that users will interact with the most. We therefore sought clarity as to whether the design and layout of the upload page was intuitive and easy to use. Again, the overwhelming majority of responses found this to be the case.



7. Finally, we had a feedback section where respondents could provide us with any thoughts or improvements they might have. There was mostly positive feedback, as well as a few suggestions for features and improvements. One such comment can be seen below, which we ended up taking on board and implementing in Fileflo.



8.4 API Testing

Below is a series of screenshots of all the API Testing we implemented. API testing ensured the AWS Lambda functions were getting and posting the correct information back and forth between a user and the API Endpoint in API Gateway.

Login API

The tests shown below make HTTP POST requests to the API endpoint. There are four tests that check for different scenarios: when required fields are missing, when a user does not

exist, when the password is incorrect, and when a user is successfully logged in. Each test sends a POST request with different parameters and expects a certain response code and message.

```
 PASS  ./login.test.js
  Lambda Function Login API Tests
    ✓ should return 401 when required fields are missing (972 ms)
    ✓ should return 403 when a user does not exist (520 ms)
    ✓ should return 403 when the password is incorrect (913 ms)
    ✓ should return 200 when a user is successfully logged in (424 ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        4.545 s
Ran all test suites matching /login.test.js/i.
```

Enter Passphrase API

The tests shown below send HTTP requests to the Lambda Function API endpoints and check the returned HTTP status codes and messages. The first test verifies that the API returns a 401 status code when the required fields are missing. The second test checks that a 404 status code is returned when the API is given to a non-existent user. The third test ensures that a 200 status code is returned when a passphrase is successfully added for an existing user.

```
└─> jest enterPassphrase.test.js
 PASS  ./enterPassphrase.test.js
  Lambda Function Folder Passphrase API Tests
    ✓ should return 401 when required fields are missing (937 ms)
    ✓ should return 404 when a user is not found (611 ms)
    ✓ should return 200 when a passphrase is successfully added (379 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        3.481 s
Ran all test suites matching /enterPassphrase.test.js/i.
```

Folder Passphrase Check API

```
└─> jest folderPassphraseCheck.test.j
 PASS  ./folderPassphraseCheck.test.js
  Lambda Function Folder Passphrase Check API Tests
    ✓ should return 401 when required fields are missing (234 ms)
    ✓ should return 404 when a user is not found (251 ms)
    ✓ should return 200 when a passphrase is successfully checked (441 ms

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        2.189 s, estimated 6 s
Ran all test suites matching /folderPassphraseCheck.test.j/i.
```

Folder Passphrase API

The tests shown below send POST requests to the Folder Passphrase API with different inputs and check the expected responses. The tests check for expected HTTP status codes and response messages when required fields are missing, when the user is not found, and when a passphrase is successfully added. The tests ensure that the API behaves correctly under different scenarios.

```
└─> jest folderPassphrase.test.js
 PASS  ./folderPassphrase.test.js
  Lambda Function Folder Passphrase API Tests
    ✓ should return 401 when required fields are missing (799 ms)
    ✓ should return 404 when a user is not found (814 ms)
    ✓ should return 200 when a passphrase is successfully added (633 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        3.798 s
Ran all test suites matching /folderPassphrase.test.js/i.
```

Confirm Passphrase API

The tests shown below are for a Lambda function called 'ConfirmPass' which confirms the passphrase for a user, and the API has four tests in total. The first test verifies that the API returns an HTTP status code of 401 if the required fields are missing. The second test confirms that the API returns a status code of 403 if the provided passphrase is incorrect. The third test checks if the API returns a status code of 404 if the user is not found. Finally, the fourth test verifies that the API returns a status code of 200 if the passphrase is correct.

```
 PASS  ./confirmPassphrase.test.js
  Lambda Function ConfirmPass API Tests
    ✓ should return 401 when required fields are missing (882 ms)
    ✓ should return 403 when passphrase is incorrect (1382 ms)
    ✓ should return 404 when user is not found (272 ms)
    ✓ should return 200 when passphrase is correct (448 ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        4.822 s, estimated 6 s
Ran all test suites matching /confirmPassphrase.test.js/i.
```

Retrieve User Private Key API

There are three of these tests: the first one checks for a missing username and expects a 400 status code with an error message. The second test checks for a non-existent user and expects a 404 status code with an error message. The third test checks for an existing user and expects a 200 status code with a response that includes the user's private key.

```
└─> jest RetrieveUserPrivateKey.test.js
 PASS  ./RetrieveUserPrivateKey.test.js
  Lambda Function Retrieve User Private Key API Tests
    ✓ should return 400 when username is missing (797 ms)
    ✓ should return 404 when a user is not found (712 ms)
    ✓ should return 200 when a user private key is successfully retrieved

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        3.282 s
Ran all test suites matching /RetrieveUserPrivateKey.test.js/i.
```

Store Private Key API

These tests are for Lambda Function Store Private Key API, which allows users to store their private key. The first test ensures that a 401 error is returned when the privateKey is missing, and the second test ensures that a 403 error is returned when the user does not exist. The third test checks that a 200 status is returned when the privateKey is successfully stored. This test assumes that there's an existing user with the specified username in the DynamoDB table.

```
└─× jest StorePrivateKey.test.js
 PASS  ./StorePrivateKey.test.js
  Lambda Function Store Private Key API Tests
    ✓ should return 401 when privateKey is missing (1009 ms)
    ✓ should return 403 when user does not exist (499 ms)
    ✓ should return 200 when privateKey is successfully stored (206 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        3.287 s
Ran all test suites matching /StorePrivateKey.test.js/i.
```

Register API

Unfortunately, the test "should return 401 when a username already exists" continues to fail despite numerous attempts to modify the Lambda function, API endpoint, and test file. However, it's worth noting that this issue does not seem to have a significant impact on the front-end user experience, as an error message is displayed indicating that the user was "not found".

```
└─× jest register.test.js
 FAIL  ./Register.test.js
  Lambda Function Register API Tests
    ✓ should return 401 when required fields are missing (293 ms)
    × should return 401 when a username already exists (124 ms)
    ✓ should return 200 when a user is successfully registered (613 ms)

  ● Lambda Function Register API Tests › should return 401 when a username already exists

    AxiosError: Request failed with status code 401

      at settle (node_modules/axios/lib/core/settle.js:19:12)
      at IncomingMessage.handleStreamEnd (node_modules/axios/lib/adapters/http.js:556:11)

Test Suites: 1 failed, 1 total
Tests:       1 failed, 2 passed, 3 total
Snapshots:   0 total
Time:        2.707 s, estimated 4 s
Ran all test suites matching /register.test.js/i.
```

Note: Register API test has a result of ⅔.

8.5 Unit Testing

During the development of Fileflo, we performed unit testing to ensure that each component rendered properly and met our expected specifications. The tests showed that the components were rendering correctly and that the various features of the application, such as file upload, file download, and folder creation, were functioning as expected. By performing unit testing, we were able to catch and fix any bugs or issues early in the

development process, which ultimately saved time and resources in the long run. Additionally, the unit tests provided us with a level of confidence that the application was functioning properly, even as we made changes and added new features over time. This allowed us to iterate and improve the application with greater efficiency and confidence. Overall, the unit testing we performed played an important role in the development of Fileflo, helping us to create a robust and reliable application for our users. Examples of our unit tests can be seen below:

App

```
PASS src/test/components/App.test.js (14.343 s)
  √ renders Fileflo link (407 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        16.113 s, estimated 26 s
Ran all test suites.
```

File Upload

```
PASS src/test/components/Upload.test.js (12.328 s)
  √ renders Upload a File title (451 ms)
  √ renders File Upload input (77 ms)
  √ renders Upload Name input (75 ms)
  √ renders File Description input (65 ms)
  √ renders Passphrase input (81 ms)

Test Suites: 1 passed, 1 total
Tests:       5 passed, 5 total
Snapshots:   0 total
Time:        13.538 s, estimated 25 s
Ran all test suites.
```

File Download

```
PASS src/test/components/Download.test.js (15.267 s)
  Form
    √ renders Form component (429 ms)
    √ renders the correct elements (234 ms)
    √ shows an error message if passphrase is empty (143 ms)
    √ submitHandler is called when form is submitted (113 ms)
    √ error message is displayed when passphrase is empty (319 ms)
```

Folder Creation

```
PASS src/test/components/Folder.test.js (11.382 s)
  Form Component
    √ renders form correctly (482 ms)
    √ folder passphrase input appears when checkbox is checked (88 ms)
    √ folder passphrase input disappears when checkbox is unchecked (92 ms)
```

8.6 Integration Testing

Our integration tests focused on testing the functionality of different components within the Fileflo application. The tests involve checking if the components are fully working, and if they meet the requirements of the user. In one of the tests, the login form is tested with valid credentials to ensure that it successfully logs in the user. The test checks if the user's session is set and if the login function is called with the correct parameters. In another test, the confirmation form is tested to ensure that it confirms the user's verification code successfully. The test checks if the verification code is entered correctly and if the confirmation function is called with the correct parameters. Other tests involve testing the functionality of components such as the Home page, About page, and the Private Key storage and retrieval API. The tests focus on ensuring that these components function as expected, and that they meet the user's requirements. The integration tests can be seen below:

Login

```
joao@ubuntu in src/test/Integration on  master [$!?] via  v14.17.6 on ☁  (eu-west-1)
× npm test -- Login.test.js

> fileflo@0.1.0 test /home/joao/Documents/College/Final_Year/Project/2023-ca400-pereirj2-cleara28/src/fileflo
> jest "Login.test.js"

 PASS  src/test/API/Login.test.js
 PASS  src/test/Integration/Login.test.js (6.144 s)

Test Suites: 2 passed, 2 total
Tests:       6 passed, 6 total
Snapshots:   0 total
Time:        7.178 s, estimated 9 s
Ran all test suites matching /Login.test.js/i.
```

Home

```
└─× npm test -- Home.test.js

> fileflo@0.1.0 test /home/joao/Documents/College/Final_Year/Project/2023-ca400-pereirj2-cleara28/src/fileflo
> jest "Home.test.js"

 PASS  src/test/Integration/Home.test.js
  Home component
    ✓ renders the home page elements (119 ms)

Test Suites: 1 passed, 1 total
Tests:       1 passed, 1 total
Snapshots:   0 total
Time:        2.181 s, estimated 3 s
Ran all test suites matching /Home.test.js/i.
```

Confirm Sign Up

```
> fileflo@0.1.0 test /home/joao/Documents/College/Final_Year/Project/2023-ca400-pereirj2-cleara28/src/fileflo
> jest "ConfirmSignUp.test.js"

 PASS  src/test/Integration/ConfirmSignUp.test.js (7.293 s)
  ConfirmSignup component
    ✓ renders the confirmation form (206 ms)
    ✓ submits the confirmation form with valid verification code (71 ms)
    ✓ shows an error message if the verification code is empty (56 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        8.286 s
Ran all test suites matching /ConfirmSignUp.test.js/i.
```