Sprint 3 Code Review #2

Overview

- Review number：#2
- Review date: 15/5
- Review name: Sprint 3 ends code review
- Review tool used: GitHub pull request open ai code review actions
- Participants: Haoyang Zheng, Yue Zhang, Naixin Xu, Junye Zhou, Yinkai Chai, Haitian Li

Selection criteria

Reviewed code for newly added functionality. Including the code to create a local database, the code to implement the table, and the Python server code for format conversion.

Goal

Ensure that the Python server can efficiently process images, ensure the implementation of local database and list display functions, ensure code quality, and ensure that adding new services does not have a potential impact on previous functions.

Scope of review:

1. Imagetransform.py
   Check whether the server handles image conversion requests efficiently
2. thermal.tsx,
   Check whether new features have any potential impact on other existing features and ensure that lists and local databases are implemented correctly

Review process

• The code awaiting review is submitted on the branch "feature: display, request, download"
• The code writer initiates a pull request and requests to be merged into the main branch
• The reviewer will check the pull request on github and compare the changed content of the file with the new content

•Run the code to verify that the function is implemented correctly
• After all reviewers determine that the code meets the purpose of code review, the code will be merged into the master branch

Review results

• Total number of issues discovered: 1
• Severity: Moderate.
• Problem description: The code of the thermal.tsx file is a little confusing, and all functions are in the same file, resulting in reduced readability.

Status: Solving

Solution: Since the demo day is coming soon, we don’t want this change to affect the Demo, so the functionality will be modularized and distributed into different files in the next sprint.

Feedback from group

1，Generally speaking, not many problems were found. After testing, all expected functions have been implemented, meeting all development expectations of sprint3. Code readability needs to be improved in the next sprint. Agree to merge into the main branch

src/PythonServer/Imagetransform.py

```python
from flask import Flask, request, send_file, jsonify
from flask_cors import CORS
import numpy as np
from PIL import Image
import io
import base64
import os

app = Flask(__name__)
CORS(app)  # Allow cross-domain requests from all origins

@app.route('/upload_image_RGB', methods=['POST'])
def upload_image_RGB():
    data = request.json
    if 'image' not in data:
        return jsonify({'error': 'No image provided'}), 400

    #Decode Base64 image
    image_data = data['image']
    if ',' in image_data:
        image_data = image_data.split(',')[1]  # remove prefix

    image_data = base64.b64decode(image_data)
    image = Image.open(io.BytesIO(image_data)).convert('RGB')

    # Convert to NumPy array
    rgb_matrix = np.array(image)

    # Save as CSV file
    csv_path = 'rgb_matrix.csv'
    np.savetxt(csv_path, rgb_matrix.reshape(-1, 3), delimiter=',', fmt='%d')

    return send_file(csv_path, as_attachment=True, download_name='rgb_matrix.csv')

def fft(image):
    # Fast Fourier transform
    f = np.fft.fft2(image)

    # Shift the low frequency component to the center
    f = np.fft.fftshift(f)

    # Fourier phase and magnitude
    magnitude = np.abs(f)
    phase = np.angle(f)

    return magnitude, phase

@app.route('/upload_image_frequency', methods=['POST'])
def upload_image_frequency():
    data = request.json
    if 'image' not in data:
        return jsonify({'error': 'No image provided'}), 400

    # Decode Base64 image
    image_data = data['image']
    if ',' in image_data:
        image_data = image_data.split(',')[1]  # remove prefix

```

```
    image_data = base64.b64decode(image_data)
    image = Image.open(io.BytesIO(image_data)).convert('L')  # Convert to grayscale image

    # Convert to NumPy array
    gray_matrix = np.array(image)

    # Perform FFT
    magnitude, phase = fft(gray_matrix)

    # Save frequency domain data as CSV file
    csv_path = 'frequency_matrix.csv'
    np.savetxt(csv_path, magnitude.reshape(-1), delimiter=',', fmt='%f')

    return send_file(csv_path, as_attachment=True, download_name='frequency_matrix.csv')



if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

src/Farmbot-Web-App-staging/frontend/thermal/thermal.tsx

```
import React, { useState, useEffect } from 'react';
import { t } from '../i18next_wrapper';
import {
  DesignerPanel,
  DesignerPanelContent,
} from '../farm_designer/designer_panel';
import { Panel } from '../farm_designer/panel_header';
import './thermal.css';
import { API } from '../api/index';

export function Thermal() {
  const [imageSrc, setImageSrc] = useState('');
  const refreshInterval = 3000;
  const [imageUrl, setImageUrl] = useState('');
  const imageUrlBase = 'http://203.101.230.232:6001/api/image/current?imgformat=JPEG';
  const [isFetching, setIsFetching] = useState(false);

  //indexDB
  const dbName = "ThermalImages";
  const storeName = "Images";

  function initDB() {
    return new Promise((resolve, reject) => {
      const request = window.indexedDB.open(dbName, 1);

      request.onerror = function (event) {
        console.error("Database error:", event.target.error);
        reject(event.target.error);
      };

      request.onupgradeneeded = function (event) {
        const db = event.target.result;
        if (!db.objectStoreNames.contains(storeName)) {
          db.createObjectStore(storeName, { keyPath: 'id' });
        }
```

```
      };

      request.onsuccess = function (event) {
        resolve(event.target.result);
      };
    });
  }

  async function storeImage(id, base64data) {
    const db = await initDB();

    return new Promise((resolve, reject) => {
      const transaction = db.transaction([storeName], "readwrite");
      const store = transaction.objectStore(storeName);
      const request = store.put({ id, base64data });


      request.onsuccess = function () {
        resolve(request.result);
      };

      request.onerror = function (event) {
        reject(event.target.error);
      };
    });
  }


  async function retrieveImage(id) {
    const db = await initDB();

    return new Promise((resolve, reject) => {
      const transaction = db.transaction([storeName], "readonly");
      const store = transaction.objectStore(storeName);
      const request = store.get(id);

      request.onsuccess = function () {
        if (request.result) {
          resolve(request.result.blob);
        } else {
          resolve(null);
        }
      };

      request.onerror = function (event) {
        reject(event.target.error);
      };
    });
  }

  async function clearDB() {
    const db = await initDB();

    return new Promise((resolve, reject) => {
      const transaction = db.transaction([storeName], "readwrite");
      const store = transaction.objectStore(storeName);
      const request = store.clear();

```

```
      request.onsuccess = function () {
        resolve();
      };

      request.onerror = function (event) {
        reject(event.target.error);
      };
    });
  }
  const handleClearDB = async () => {
    const userConfirmed = window.confirm("Are you sure you want to clear all data?");
    if (userConfirmed) {
      await clearDB();
      setTableData([]); // Clear the table data
      alert("All data has been cleared.");
    }
  };
  //////
  const [tableData, setTableData] = useState([]);

  // Fetch a single image on component mount
  useEffect(() => {
    const fetchImage = () => {
      const uniqueSuffix = new Date().getTime();
      const newUrl = `${imageUrlBase}&t=${uniqueSuffix}`;
      setImageUrl(newUrl);
      fetchAndStoreImage();
    };

    fetchImage(); // Fetch the image only once
  }, []);

  useEffect(() => {
    let intervalId = null;
    if (isFetching) {
      fetchAndStoreImage(); // Get and store the first image
      intervalId = setInterval(fetchAndStoreImage, refreshInterval); // Set timer to periodically grab and stor
    }

    return () => {
      if (intervalId) {
        clearInterval(intervalId); // clear timer
      }
    };
  }, [isFetching]); // Rerun the effect when isFetching changes

  const toggleFetching = () => {
    setIsFetching(!isFetching); // Switch crawling status
  };

  const fetchImageBlob = async () => {
    return new Promise((resolve, reject) => {
      const url = API.current.getImage
      const xhr = new XMLHttpRequest()
      xhr.open('GET', url, true);
      xhr.responseType = 'blob';
      console.log(xhr);

```

```
      xhr.onload = function () {
        if (this.status === 200) {
          resolve(this.response);
        } else {
          reject(new Error('Failed to fetch image'));
        }
      };

      xhr.onerror = function () {
        reject(new Error('Network error'));
      };

      xhr.send();
    });
  };

  const fetchAndStoreImage = async () => {
    try {
      const uniqueSuffix = new Date().getTime();
      const blob = await fetchImageBlob();

      //After requesting the image, convert it to base64 and save it in db
      const reader = new FileReader();
      reader.readAsDataURL(blob);
      reader.onloadend = async () => {
        const base64data = reader.result;

        const imageId = 'image_' + uniqueSuffix;
        await storeImage(imageId, base64data);
        setImageSrc(base64data);
        setTableData(prevData => [...prevData, {
          id: imageId,
          timestamp: new Date(uniqueSuffix).toLocaleString(),
          imageUrl: base64data
        }]);
      };
    } catch (error) {
      console.error('Error fetching and storing image:', error);
    }
  };

  //Create a new function to load all images from IndexedDB
  async function loadImagesFromDB() {
    const db = await initDB();

    return new Promise((resolve, reject) => {
      const transaction = db.transaction([storeName], "readonly");
      const store = transaction.objectStore(storeName);
      const request = store.getAll();

      request.onsuccess = function () {
        resolve(request.result);
      };

      request.onerror = function (event) {
        reject(event.target.error);
      };
    });
```

```
  }

  //Initialize table data.
  useEffect(() => {
    loadImagesFromDB().then(images => {
      setTableData(images.map(img => ({
        id: img.id,
        timestamp: new Date(parseInt(img.id.replace('image_', ''))).toLocaleString(),
        imageUrl: img.base64data
      })));
    }).catch(err => console.error("Failed to load images from DB:", err));
  }, []);


  //Download source image
  const handleDownload = async (base64Image, fileName = 'thermal_image.jpeg') => {
    if (!base64Image) return;

    const link = document.createElement('a');
    link.href = base64Image;
    link.download = fileName;
    document.body.appendChild(link);
    link.click();
    document.body.removeChild(link);
  };

// Send Base64 image to Python server and handle CSV file download
const handleDownloadRGBMatrix = async (base64Image, timestamp) => {
  try {
    const response = await fetch("http://203.101.230.232:5000/upload_image_RGB", {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({ image: base64Image }),
    });
    console.log(response);

    if (response.ok) {
      const blob = await response.blob();
      const url = window.URL.createObjectURL(blob);
      const a = document.createElement('a');
      a.style.display = 'none';
      a.href = url;
      a.download = `${timestamp}_rgb_matrix.csv`;
      document.body.appendChild(a);
      a.click();
      window.URL.revokeObjectURL(url);
    } else {
      console.error('Failed to download CSV');
    }
  } catch (error) {
    console.error('Error downloading CSV:', error);
  }
};

// Send Base64 image to Python server and handle CSV file download
const handleDownloadFrequencyMatrix = async (base64Image, timestamp) => {
```

```
  try {
    const response = await fetch("http://203.101.230.232:5000/upload_image_frequency", {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({ image: base64Image }),
    });
    console.log(response);

    if (response.ok) {
      const blob = await response.blob();
      const url = window.URL.createObjectURL(blob);
      const a = document.createElement('a');
      a.style.display = 'none';
      a.href = url;
      a.download = `${timestamp}_frequency_matrix.csv`;
      document.body.appendChild(a);
      a.click();
      window.URL.revokeObjectURL(url);
    } else {
      console.error('Failed to download CSV');
    }
  } catch (error) {
    console.error('Error downloading CSV:', error);
  }
};

  return (
    <DesignerPanel panelName="thermal" panel={Panel.Thermal}>
      <DesignerPanelContent panelName="thermal">
        <label>{t('Thermal Camera User Web')}</label>

        <div>
          <iframe
            src="http://203.101.230.232:6001/"
            title="Embedded Page"
            id="iframe"
          />
        </div>
        <label>{t('real-time capture')}</label>
        <div className="image-container">
          <img src={imageSrc} alt="Loading...." />
        </div>

        <button
          className={`request-button ${isFetching ? 'stop' : ''}`}
          onClick={toggleFetching}
        >
          {isFetching ? 'stop' : 'start take photo'}{' '}
        </button>
        <button id="clear_button" onClick={handleClearDB}>Clear Images</button>
        <table className="data-table">
          <thead>
            <tr>
              <th>{t('Timestamp')}</th>
              <th>{t('Image')}</th>
              <th>{t('Format')}</th>
```

```
            </tr>
          </thead>
          <tbody>
            {tableData.map((row) => (
              <tr key={row.id}>
                <td>{row.timestamp}</td>
                <td>
                  <img src={row.imageUrl} alt="Request Error" style={{ width: '100px', height: 'auto' }} />
                </td>                <td>
                <button onClick={() => handleDownload(row.imageUrl, `${row.timestamp}_Image_${row.id}.jpeg`)}>D
                <button onClick={() => handleDownloadRGBMatrix(row.imageUrl, row.timestamp)}>Download RGB Matri
                <button onClick={() => handleDownloadFrequencyMatrix(row.imageUrl, row.timestamp)}>Download Fre
                </td>
              </tr>
            ))}
          </tbody>
        </table>


      </DesignerPanelContent>
    </DesignerPanel>
  );
}
```

Feedback from AI

src/Farmbot-Web-App-staging/frontend/thermal/thermal.tsx
The code provided is a React component for handling thermal images, storing them in IndexedDB, and providing functionality for downloading images and data matrices. Below is an evaluation based on the provided checklist:

1. **Documentation Defects:**
   - **Naming:**
     - Most variable and function names are clear and descriptive (`initDB`, `storeImage`, `fetchImageBlob`). However, names like `t` imported from `i18next_wrapper` could be more descriptive.
   - **Comment:**
     - Comments are somewhat sparse and could be more descriptive in explaining the purpose and functionality of blocks of code, especially complex functions.
2. **Visual Representation Defects:**
   - **Bracket Usage:**
     - Brackets are used correctly throughout the code.
   - **Indentation:**
     - Indentation is generally consistent and follows common React/JavaScript practices, aiding readability.
   - **Long Line:**
     - Some lines are quite long, particularly URL constructions and function calls with multiple parameters. These could be broken into multiple lines for better readability.
3. **Structure Defects:**
   - **Dead Code:**

    - No obvious dead code detected in the snippet provided.
  - **Duplication:**
    - The functions `handleDownloadRGBMatrix` and `handleDownloadFrequencyMatrix` are very similar and could potentially be refactored into a single function that accepts parameters to handle different types of downloads.
4. **New Functionality:**
  - **Use Standard Method:**
    - The use of `XMLHttpRequest` could be replaced with the Fetch API, which is more modern and generally easier to work with in React applications.
5. **Resource Defects:**
  - **Variable Initialization:**
    - All variables are initialized appropriately.
  - **Memory Management:**
    - Proper cleanup is performed in `useEffect` hooks with return functions to clear intervals, which is good for memory management.
6. **Check Defects:**
  - **Check User Input:**
    - There is a basic check with `window.confirm` before clearing the database which is appropriate. More rigorous input validation might be needed depending on further requirements.
7. **Interface Defects:**
  - **Parameter:**
    - Parameters in function calls are correctly used.
8. **Logic Defects:**
  - **Compute:**
    - Logic appears correct for the intended functionality of the component.
  - **Performance:**
    - The performance could potentially be impacted by frequent database operations and the use of synchronous `await` inside loops or iterative operations could be optimized.

Summary of Recommendations:

- Improve comments to better describe the functionality.
- Consider breaking long lines into multiple lines.
- Refactor similar functions (`handleDownloadRGBMatrix` and `handleDownloadFrequencyMatrix`) to reduce code duplication.
- Replace `XMLHttpRequest` with the Fetch API for fetching image data.
- Consider potential performance optimizations around database and network operations, especially if operating at scale.

These changes should improve the maintainability, readability, and performance of the code.

---

src/PythonServer/Imagetransform.py

Documentation Defects:

- **Naming:**
  - Variable names like `data` , `image_data` , and `csv_path` are clear and descriptive.
  - Function names such as `upload_image_RGB` and `upload_image_frequency` clearly describe their functionality.
- **Comment:**
  - Comments are minimal. Adding comments to explain the purpose of each function, especially complex operations like the FFT process, would enhance clarity.

Visual Representation Defects:

- **Bracket Usage:**
  - Brackets are used correctly throughout the code.
- **Indentation:**
  - Indentation is consistent, making the code readable.
- **Long Line:**
  - The lines of code are not excessively long, maintaining readability.

Structure Defects:

- **Dead Code:**
  - No apparent dead code is present.
- **Duplication:**
  - There is no significant duplication of code, although the image decoding process could be refactored into a helper function.

New Functionality:

- **Use Standard Method:**
  - The methods used for handling image uploads and processing are standard in Python and Flask.

Resource Defects:

- **Variable Initialization:**
  - All variables are correctly initialized.
- **Memory Management:**
  - Proper memory management is in place, but consider cleaning up temporary files after sending them.

Check Defects:

- **Check User Input:**
  - The code checks for the presence of the 'image' key in the input data but could include additional validation for the image format and size.

Interface Defects:

- **Parameter:**
  - No parameters are missing or incorrect.

Logic Defects:

- **Compute:**
  - The logic for handling images and performing FFT is correctly implemented.
- **Performance:**
  - The performance is adequate, but consider handling larger images in a more memory-efficient way.

Summary of Recommendations:

1. **Add Comments:** Provide comments to explain the purpose of each function and complex operations.
2. **Refactor Image Decoding:** Refactor the image decoding process into a helper function to avoid duplication.
3. **Clean Up Temporary Files:** Ensure temporary files like CSVs are cleaned up after sending them to the client.
4. **Enhance Input Validation:** Add additional validation for the image format and size.