Amir Farooq
Ian Keilman
Peyton Pepkowski
Dr. Greg Matthews
STAT 370-01E
2 May 2025

Data Science Consulting Project Report: Thermal Imaging

Introduction

This report outlines the development of a comprehensive database designed to link thermal image URLs to their corresponding metadata through a unique identifier. The thermal images and associated data were originally collected on the windows and doors of Cuneo and Dumbach Hall on the Loyola University Chicago Lakeshore Campus during the Fall 2024 semester by a group of four graduate students. While the original project included statistical analyses conducted by the students to explore specific research questions regarding the energy efficiency and structural performance of the buildings, these analyses are not utilized in the current work. Instead, this project focuses on repurposing the collected thermal images and their logistical metadata to build a structured and searchable dataset. Additionally, a photoshop automation algorithm was developed to systematically remove embedded data from the thermal images themselves, ensuring a clean and consistent visual. This system not only organizes the existing data but also establishes a framework that allows new thermal images and their metadata to be easily incorporated and accurately linked within the database.

Data

The dataset compiled for this project consists of thermal images, each accompanied by a set of metadata detailing the conditions and context where the photo was captured. All information is stored in a Google Drive folder. For every image, a photo number and a unique photo ID are recorded, serving as identifiers corresponding to the image file. Additional logistical information includes the building name and the specific side of the building where the image was taken (e.g., north, south, east, or west). Geographic coordinates are included through both latitude and longitude values, enabling precise spatial referencing. Temporal data comprises the exact date and time the image was captured. Each thermal image is further annotated with observed temperature values, as well as the minimum and maximum temperatures detected in the image frame (all in degrees Celsius). The framing quality of each image is also documented, indicating whether the photo was correctly framed on a window, door, an open window, or if the framing was improper, such as if the whole window was not fully in the photo. All images were taken at a fixed distance of 10 meters from the target door or window to ensure consistency. Environmental context is provided through the recorded outdoor temperature (in Celsius) at the time of capture and the direction the sun was facing. Furthermore, the dataset includes whether the image was taken from the interior or exterior of the building, and the floor level - categorized as ground level, second floor, third floor, or fourth floor - on which the image was captured. This comprehensive metadata structure provides a foundation for organizing, analyzing, and expanding the database and allows for abundant details to be defined for each image.

Methods

Photoshop Algorithm

This script was written in python. It contains two parts, extracting minimum and maximum temperatures from the photo's overlay, then getting rid of the overlay to improve machine learning performance. Extracting min/max temps are crucial to allowing ML solutions to work as temp to color keys are dynamic and change in every image. Taking the temps allows us to understand which colors represent which temperatures in each image.

The min/max extraction is done using an open source text reading NN, pytesseract. The numbers appear in the same spot over every image, so a bounding box was set for each number's location to allow for the OCR model to focus on each reading and reduce the amount of noise imputed to the model. Each min and max appears twice, so the OCR model does four readings and chooses the highest confidence out of the two readings for each. This lets us have a higher confidence as opposed to only reading the temperatures from one location in each image. Also, if the confidence for a reading is below a threshold, – Chosen to be 80% – the user will be asked to check the image themselves to avoid errors. This automation allows the user to save time when uploading images, while preserving the accuracy needed for analysis.

The actual photoshop is not really photoshop. We take advantage of the fact that our desired overlay to remove appears in the same areas in each image. The overlay includes readings in all four corners and a temperature bar on the right side. We want to remove the overlay and replace it with an estimate of what may have been there. We do this by detecting pixels with a high whiteness value, and setting these pixel locations to a "white mask." We also add all pixels within a 2 pixel radius of any white pixel to this mask. This helps with some outlines not detected in the initial scan. All other pixels are stored in the inverse of this. For each pixel in white mask, the nearest 150 pixels not in the white mask are averaged and the white pixel is replaced. This process is also done for every pixel in the temperature bar. We perform this smoothing twice to make sure everything gets smoothed and the image is then ready for analysis. If maximum performance was desired for the photoshop algorithm, some sort of generative AI may work, however the training data needed as well as time spent learning about and creating that model make this option much less ideal than our solution.

Database Creation

In constructing the thermal image database, we employed Python and the sqlite3 package to build a local SQLite database and define its schema. We organized the database in a multi-table relational model and composed SQL statements in a Python script to set up its schema. We also added PRAGMA foreign_keys = ON to implement referential integrity and only created all of the tables if they didn't already exist. This cleanly initialized the database and made it ready to be populated with data. After setting up the schema, we utilized individual functions written in Python to automate populating the database from two main sources: a shared Google Drive and a centralized Google Spreadsheet.

In populating the database with image data, we employed the Google Drive API to access programmatically particular Drive folders with thermal image files. To authenticate access, we used a service account and generated public viewable URLs for all images. Filenames were parsed and standardized to pull consistent location_id values such that images were distinctly associated with metadata. These standardized IDs were important in joining data between tables. The filename, date, and generated URL of every image were inserted into the database through parameterized SQL statements.

We utilized the Google Sheets API (through the gspread library) to access a Google Sheet named ThermalData. We read in all the records as dictionaries and parsed through each row to retrieve and clean key information like building name, floor level, time taken, and temperature readings. We introduced logic to convert time-stamps to a uniform format and to categorize floor levels and window status in a uniform manner. We inserted this clean and formatted data into relevant database tables through SQL, associating each entry with data to its corresponding image through the shared location_id. Error checking was also put in place to circumvent any erroneous entries like incorrectly formatted time-stamps or bad photo IDs.

This combined strategy, constructing the database schema ab initio and filling it programmatically through APIs, enabled a resilient, repeatable, and scalable pipeline to map thermal image data with environmental context information. Database operations were all done through sqlite3, and the entire pipeline ends with committing all transactions and closing out to guarantee data integrity.

Results

The resultant is a fully functional relational database known as thermal_img.db that consolidates thermal image data, location metadata for windows, and environmental readings. It includes three primary tables: static, images, and environment_logs, all of which are relationally connected through a location_id. The database was written to be both human-readable and machine-querable and is therefore suitable for downstream processing in additional analysis, visualization platforms, or automated pipelines.

The static table is the root of the database and contains principal metadata for a particular location. This entails a unique location_id, building name (cuneo_hall or dumbach_hall), floor number (in a numerical form), and building side (west, east, south, or north). The table ensures every location is defined in a unique way and can be used by other tables. The images table stores data about every thermal image file gathered from Google Drive. The entry for each includes a unique image_id, the location_id associated with it, the image file name, the date of capture, and a publicly available URL linking directly to the image in Google Drive. The links were generated with the Drive API and may be used to view or download the images in other programs. The environment_logs table contains environmental readings like external temperature, minimum and maximum temperature at the window, hour of image taken, windows opened (y/n), and measurement date. Similar to the images table, every row is linked to a location_id and also shares the same image URL for easy access and visualization purposes.

Since all tables are linked with location_id, it is simple to combine data from several tables by joining them in a simple SQL query. For instance, to see building name, max_temp, and image_path of all west windows, you could type:

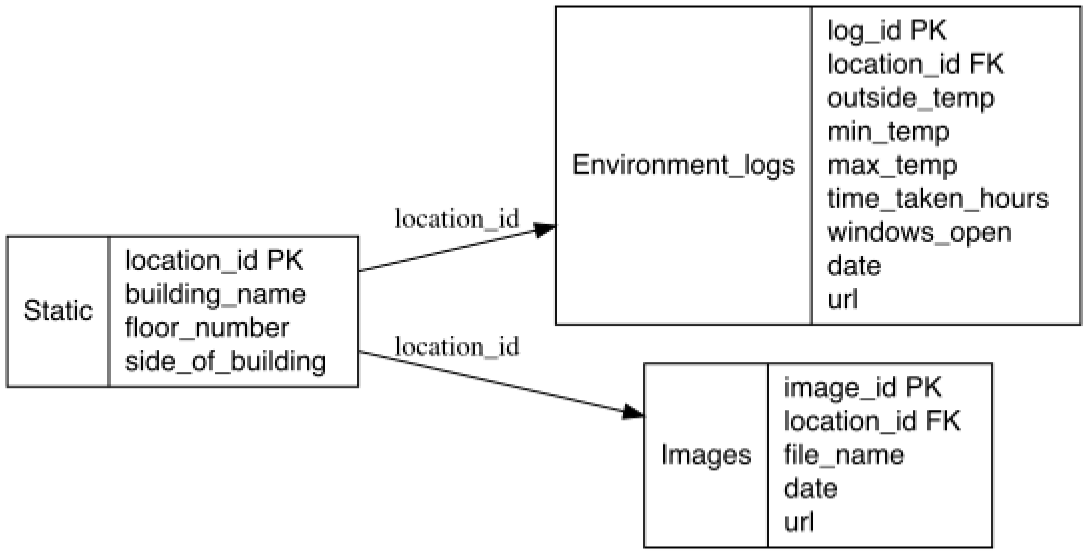
```
SELECT s.building_name, e.max_temp, i.url
FROM environment_logs e
JOIN Static s ON s.location_id = e.location_id
JOIN images i ON e.location_id = i.location_id
WHERE s.side_of_building = west
```

This request illustrates how data in all three of these tables can be easily combined to yield useful insights. Analysts can equally request temperature trends by building, floor-by-floor comparisons, or isolate readings for windows with open vs. closed states.

The database is designed to be reusable and scalable. Since data is organized in logical tables and joined through foreign keys, new measurements or images may be easily introduced without causing much disturbance. Additionally, because image URLs are saved directly, the database acts as a light index to larger image files saved elsewhere in Google Drive.

Figure 1 shows an entity relationship diagram depicting the relationship between each table and its respective data. In the diagram, ‘PK’ represents a primary key; a primary key uniquely identifies each record in an entity set. In our database, log_id is the primary key for ‘environment_logs’, location_id is the primary key for ‘static’, and image_id is the primary key for ‘images.’ ‘FK’ represents a foreign key, which consists of an attribute in one entity that references the primary key of another entity, creating a relationship between the two. In the ‘environment_logs’ table and the ‘images’ table, location_id is a foreign key because they each reference the ‘static’ table.

Figure 1



In short, the resultant database is a thorough representation of window-level thermal data, well-normalized for efficiency, and allows flexible querying and analysis by means of SQL. It is appropriate to utilize in building efficiency studies, anomaly detection, etc.

Conclusion

In this project, we developed a structured and scalable relational database that connects thermal image URLs to their corresponding metadata using a unified identifier system. The process involved three major components: cleaning and preprocessing image data using a Python-based image overlay removal algorithm, organizing the thermal image metadata into a standardized format, and constructing a multi-table SQLite database to store and link all information. Images and data were collected from buildings on the Loyola University Chicago Lakeshore campus and were retrieved and processed from shared Google Drive and Sheets resources.

The result is a fully functional database that efficiently organizes images, environmental context, and metadata across three interconnected tables: ‘static,’ ‘image,’ and ‘environment_logs.’ Each entry is linked by a common location_id, allowing for SQL queries that can support further analysis, such as identifying temperature anomalies by building side, comparing floors, or tracking window conditions. The system is designed to be extensible, enabling future additions of images and data with minimal restructuring. Ultimately, this database provides a solid foundation for future research regarding thermal imagery and energy efficiency.

Future work on this project could explore advanced data analysis and modeling techniques to enhance the utility of the thermal image database. First, machine learning models could be developed to automatically detect patterns or anomalies in thermal images - such as areas of unexpected heat loss - improving efficiency in building diagnostics. The analysis of data over time is another promising direction, where thermal data collected across different seasons could reveal how insulation performance or energy efficiency varies over time, helping identify long-term trends or deterioration. Lastly, integrating this data with building energy models could provide more accurate simulations and predictions. By correlating thermal imaging with actual energy usage, researchers could better understand how surface-level heat signatures relate to overall building performance, leading to more effective reconstruction or sustainability strategies.

References

Boateng, Bernard, et al. *Thermal Imaging Analysis*. Loyola University Chicago, 2024. Unpublished manuscript.

Boateng, Bernard, et al. *Thermal Imaging Analysis.* Unpublished dataset. Loyola University Chicago. Google Drive, https://drive.google.com/drive/folders/1XTCgc30yj-4-hxexQj9oRKMnzv4HxVfx

Appendix

Files also at https://github.com/IanKeilman/370-Database/tree/main

```
import numpy as np
from PIL import Image, ImageEnhance, ImageFilter, ImageOps
import pytesseract
from scipy.ndimage import binary_dilation
from scipy.spatial import KDTree

# Configuration constants
OCR_REGIONS = {
    "min_top_left": {"x": 36,  "y": 31,  "width": 34, "height": 15},
    "min_right":    {"x": 199, "y": 261, "width": 36, "height": 19},
    "max_top_left": {"x": 35,  "y": 45,  "width": 35, "height": 17},
    "max_right":    {"x": 197, "y": 38,  "width": 38, "height": 18},
}

WHITE_TOLERANCE = 150      # how "white" a pixel must be (0–255 scale)
NEIGHBOR_COUNT = 150       # how many non-white neighbors to average

WHITE_BOXES = [
    {"x": -1,  "y": 0,   "width": 71, "height": 62},
    {"x": 166, "y": 3,   "width": 70, "height": 26},
    {"x": 197, "y": 36,  "width": 38, "height": 244},
    {"x": 177, "y": 299, "width": 54, "height": 21},
    {"x": 0,   "y": 302, "width": 46, "height": 17},
]

GREEN_BOX = {"x": 99, "y": 140, "width": 38, "height": 41}
SMOOTH_TEMPERATURE_BAR = True
TEMPERATURE_BAR_BOX = {"x": 215, "y": 56, "width": 20, "height": 209}


def run_ocr_on_coords(image_path, regions=OCR_REGIONS, conf_threshold=80):
    """
    Crop each named region, enhance contrast, sharpen, then run OCR.
    Returns a dict with 'min', 'max', their confidence scores, and flags if below threshold.
    """
    img = Image.open(image_path)
```

```
raw_results = {}

# Run OCR on every region
for label, box in regions.items():
    x, y, w, h = box["x"], box["y"], box["width"], box["height"]
    crop = img.crop((x, y, x + w, y + h))

    # Pre-process: grayscale → boost contrast → sharpen
    gray     = ImageOps.grayscale(crop)
    enhanced = ImageEnhance.Contrast(gray).enhance(2.0)
    sharp    = enhanced.filter(ImageFilter.SHARPEN)

    # Extract text data
    data = pytesseract.image_to_data(
        sharp,
        output_type=pytesseract.Output.DICT,
        config="--psm 6 -c tessedit_char_whitelist=0123456789."
    )

    # Select highest-confidence fragment
    best_text, best_conf = "", -1
    for txt, conf in zip(data["text"], data["conf"]):
        try:
            c = float(conf)
        except ValueError:
            continue
        t = txt.strip()
        if t and c > best_conf:
            best_text, best_conf = t, c
    raw_results[label] = {"value": best_text, "conf": best_conf}

# Choose best min and max
def choose(a, b):
    r1, r2 = raw_results[a], raw_results[b]
    return r1 if r1["conf"] >= r2["conf"] else r2

min_res = choose("min_top_left", "min_right")
max_res = choose("max_top_left", "max_right")

return {
        "min":      min_res["value"],
        "min_conf": min_res["conf"],
        "min_flag": min_res["conf"] < conf_threshold,
        "max":      max_res["value"],
        "max_conf": max_res["conf"],
        "max_flag": max_res["conf"] < conf_threshold,
    }


def smooth_pixels(
    image_path, output_path,
    tolerance=WHITE_TOLERANCE, neighbor_count=NEIGHBOR_COUNT,
    white_boxes=WHITE_BOXES, green_box=GREEN_BOX,
    smooth_bar=SMOOTH_TEMPERATURE_BAR, bar_box=TEMPERATURE_BAR_BOX
):
    """
    1) Find near‑white pixels (>= 255‑tolerance), dilate mask by 2px.
    2) Keep only those pixels inside white_boxes.
    3) Replace each by average of k nearest non‑white pixels (two passes).
    4) k‑NN blur for green pixels in green_box.
    5) Optionally blur temperature bar in bar_box.
    """
    img  = Image.open(image_path).convert("RGB")
    arr  = np.array(img)
    h, w = arr.shape[:2]
    white_t = 255 - tolerance

    # 1) White mask + dilation
    white_mask = np.all(arr >= white_t, axis=-1)
    struct     = np.ones((5, 5), bool)
    dilated    = binary_dilation(white_mask, structure=struct)

    # 2) Filter to white_boxes
    coords = np.argwhere(dilated)
    keep = []
    for y, x in coords:
        for b in white_boxes:
            if b["x"] <= x < b["x"]+b["width"] and b["y"] <= y < b["y"]+b["height"]:
                keep.append((y, x))
                break
```

```
white_coords = np.array(keep)
if white_coords.size == 0:
    print("No white pixels to smooth.")
    return 0

non_white = np.argwhere(~dilated)
tree      = KDTree(non_white)

smoothed = arr.copy()
changed  = 0
# 3) Two passes k-NN blur
for _ in range(2):
    for y, x in white_coords:
        dists, idxs = tree.query((y, x), k=neighbor_count)
        nbrs       = non_white[idxs]
        colors      = arr[nbrs[:,0], nbrs[:,1]]
        avg_color   = colors.mean(axis=0).astype(np.uint8)
        if not np.array_equal(smoothed[y, x], avg_color):
            smoothed[y, x] = avg_color
            changed      += 1
    arr = smoothed.copy()

# 4) Blur green pixels
gx, gy, gw, gh = green_box.values()
green_coords = []
green_box_mask = np.zeros_like(white_mask)
for i in range(gy, gy+gh):
    for j in range(gx, gx+gw):
        r, g, b = arr[i, j]
        if g > 100 and g > r and g > b:
            green_coords.append((i, j))
            green_box_mask[i, j] = True
green_coords = np.array(green_coords)
outside_coords = np.argwhere(~green_box_mask)
if outside_coords.size and green_coords.size:
    green_tree = KDTree(outside_coords)
    for y, x in green_coords:
        k = min(neighbor_count, len(outside_coords))
        _, idxs = green_tree.query((y, x), k=k)
        nbrs   = outside_coords[idxs]
```

```
            avg    = arr[nbrs[:,0], nbrs[:,1]].mean(axis=0).astype(np.uint8)
            smoothed[y, x] = avg
            changed += 1

    # 5) Blur temperature bar
    if smooth_bar:
        bx, by, bw, bh = bar_box.values()
        inside = [(y, x)
                  for y in range(by, min(by+bh, h))
                  for x in range(bx, min(bx+bw, w))]
        outside = np.argwhere(~(
            (np.arange(h)[:,None] >= by) & (np.arange(h)[:,None] < by+bh) &
            (np.arange(w)[None,:] >= bx) & (np.arange(w)[None,:] < bx+bw)
        ))
        bar_tree = KDTree(outside)
        for y, x in inside:
            _, idxs = bar_tree.query((y, x), k=min(neighbor_count, len(outside)))
            nbrs    = outside[idxs]
            avg     = smoothed[nbrs[:,0], nbrs[:,1]].mean(axis=0).astype(np.uint8)
            smoothed[y, x] = avg
            changed += 1
        print("Temperature bar smoothed.")

    Image.fromarray(smoothed).save(output_path)
    print(f"Saved smoothed image to {output_path}")
    return changed

import os
import io
import sqlite3
from PIL import Image
import pytesseract
import cleaning_funcs as utils
from cleaning_funcs import (
    run_ocr_on_coords, smooth_pixels,
    OCR_REGIONS, WHITE_TOLERANCE, NEIGHBOR_COUNT,
    WHITE_BOXES, GREEN_BOX,
    SMOOTH_TEMPERATURE_BAR, TEMPERATURE_BAR_BOX
)
from google.oauth2 import service_account
from googleapiclient.discovery import build
from googleapiclient.http import MediaIoBaseDownload, MediaFileUpload
import matplotlib.pyplot as plt

# Ensure pytesseract is configured correctly
pytesseract.pytesseract.tesseract_cmd = (
    r"C:\\Program Files\\Tesseract-OCR\\tesseract.exe"
)

# ---------- Configuration ----------
SERVICE_ACCOUNT_FILE = 'acoustic-atom-456322-b4-ff317ed933d2.json'
SCOPES               = ['https://www.googleapis.com/auth/drive']
FOLDER_ID            = '1cTzfg8extLH7V3hiXgFXgmVvUkCz4jkc'
DOWNLOAD_DIR         = 'downloaded_images'
PROCESSED_DIR        = 'processed_images'
DB_PATH              = 'thermal_img.db'

os.makedirs(DOWNLOAD_DIR, exist_ok=True)
os.makedirs(PROCESSED_DIR, exist_ok=True)

# ---------- Google Drive Authentication ----------
def authenticate_drive():
    creds = service_account.Credentials.from_service_account_file(
        SERVICE_ACCOUNT_FILE, scopes=SCOPES
    )
    return build('drive', 'v3', credentials=creds)

# ---------- Drive Utilities ----------
def list_files_in_folder(service, folder_id):
    files = {}
    page_token = None
    query = f"'{folder_id}' in parents and trashed = false"
    while True:
        resp = service.files().list(
            q=query, spaces='drive',
            fields='nextPageToken, files(id, name, mimeType)',
            pageToken=page_token
        ).execute()
        for f in resp.get('files', []):
            files[f['name']] = f['id']
        page_token = resp.get('nextPageToken')
        if not page_token:
            break
    return files

def download_file(service, file_id, destination_path):
    request = service.files().get_media(fileId=file_id)
    os.makedirs(os.path.dirname(destination_path), exist_ok=True)
    with io.FileIO(destination_path, 'wb') as fh:
        downloader = MediaIoBaseDownload(fh, request)
        done = False
        while not done:
            status, done = downloader.next_chunk()
            print(f"Downloading {os.path.basename(destination_path)}: {int(status.progress() *
100)}%")

# ---------- Main Processing ----------
if __name__ == '__main__':
    drive_service = authenticate_drive()
    remote_files  = list_files_in_folder(drive_service, FOLDER_ID)

    # Connect to SQLite
    conn   = sqlite3.connect(DB_PATH)
    cursor = conn.cursor()

    # Ensure url_original & url_clean columns exist
    cursor.execute("PRAGMA table_info(environment_logs)")
    cols = [r[1] for r in cursor.fetchall()]
    if 'url_original' not in cols:
        cursor.execute("ALTER TABLE environment_logs ADD COLUMN url_original TEXT")
    if 'url_clean' not in cols:
        cursor.execute("ALTER TABLE environment_logs ADD COLUMN url_clean TEXT")
    conn.commit()

    for name, file_id in remote_files.items():
        # skip cleaned or non-images
        if 'clean' in name.lower() or not name.lower().endswith(('.jpg','.jpeg','.png')):
            continue

        base       = os.path.splitext(name)[0]
```

```
dl_path    = os.path.join(DOWNLOAD_DIR, name)
clean_name = f"{base}_clean.jpg"
cl_path    = os.path.join(PROCESSED_DIR, clean_name)

# Download original if needed
if not os.path.exists(dl_path):
    download_file(drive_service, file_id, dl_path)

# skip if cleaned already exists remotely
if clean_name in remote_files:
    continue

# OCR step
ocr = run_ocr_on_coords(dl_path, OCR_REGIONS)
print(f"OCR for {name}: {ocr}")

# manual override if confidence < 80
if ocr['min_conf'] < 80 or ocr['max_conf'] < 80:
    img = Image.open(dl_path)
    plt.ion()
    plt.figure(figsize=(8,6))
    plt.imshow(img)
    plt.axis('off')
    plt.show()

    print(f"\n⚠️ Low OCR confidence for {name}.")
    if ocr['min_conf'] < 80 and ocr['max_conf'] < 80:
        ocr['min'] = input("Enter MIN temperature: ")
        ocr['max'] = input("Enter MAX temperature: ")
    elif ocr['min_conf'] < 80:
        ocr['min'] = input("Enter MIN temperature: ")
    else:
        ocr['max'] = input("Enter MAX temperature: ")

    plt.close()

# smoothing step
changed = smooth_pixels(
    dl_path, cl_path,
    tolerance=WHITE_TOLERANCE,
    neighbor_count=NEIGHBOR_COUNT,
    white_boxes=WHITE_BOXES,
    green_box=GREEN_BOX,
    smooth_bar=SMOOTH_TEMPERATURE_BAR,
    bar_box=TEMPERATURE_BAR_BOX
)
print(f"Pixels changed: {changed}")

# upload cleaned image
media = MediaFileUpload(cl_path, mimetype='image/jpeg')
meta  = {'name': clean_name, 'parents': [FOLDER_ID]}
up    = drive_service.files().create(
    body=meta, media_body=media, fields='id'
).execute()

url_original = f"https://drive.google.com/uc?id={file_id}"
url_clean    = f"https://drive.google.com/uc?id={up['id']}"

# ----- manual metadata prompts -----
location_id      = input("Enter location_id (e.g. ch_1_2): ").strip()
outside_temp     = float(input("Enter outside temperature (°F): "))
time_taken_hours = int(input("Enter hour of day (0–23): "))
windows_opened   = input("Window opened? (Y/N): ").strip().upper()
while windows_opened not in ('Y','N'):
    windows_opened = input("Please enter Y or N: ").strip().upper()

# check for duplicates
cursor.execute(
    "SELECT 1 FROM environment_logs WHERE url_original=? AND url_clean=?",
    (url_original, url_clean)
)
if cursor.fetchone():
    print("Already logged; skipping.")
    continue

# final insert: 8 placeholders for 8 values
cursor.execute(
    """
    INSERT INTO environment_logs
      (location_id,
                outside_temp,
                min_temp,
                max_temp,
                time_taken_hours,
                windows_opened,
                date,
                url_original,
                url_clean)
            VALUES (?, ?, ?, ?, ?, ?, date('now'), ?, ?)
            """,
            (
                location_id,
                outside_temp,
                float(ocr['min']),
                float(ocr['max']),
                time_taken_hours,
                windows_opened,
                url_original,
                url_clean
            )
        )
        conn.commit()
        print(f"Logged {name} → {clean_name}")

    conn.close()
    print("Processing complete.")


import os
import re
import sqlite3
from datetime import datetime

import gspread
from googleapiclient.discovery import build
from google.oauth2 import service_account
from oauth2client.service_account import ServiceAccountCredentials

#Configuration
DB_PATH = os.path.abspath("thermal_img.db")
```

```
CREDENTIALS_FILE = "credentials.json"
SHEET_NAME = "ThermalData"
DRIVE_FOLDER_IDS = {
  "cuneo_hall_cleaned": "1aEbEik1xkeiVpG1J6azLIYlKIgcPKLY1",
  "dumbach_hall_cleaned": "1VWy0DqUseUzjMEqjKEf4e78nxwp1t9iq"
}
DEFAULT_DATE = "2024-11-02"

#Authenticate
SCOPES = ["https://spreadsheets.google.com/feeds", "https://www.googleapis.com/auth/drive"]
creds = service_account.Credentials.from_service_account_file(CREDENTIALS_FILE,
scopes=SCOPES)
sheets_creds = ServiceAccountCredentials.from_json_keyfile_name(CREDENTIALS_FILE,
SCOPES)
sheet_client = gspread.authorize(sheets_creds)
drive_service = build('drive', 'v3', credentials=creds)

#Database connect
conn = sqlite3.connect(DB_PATH)
cursor = conn.cursor()

#Location id same for all tables
def standardize_location_id(photo_id):
  base = photo_id.strip().split(".")[0]
  match = re.match(r"([a-zA-Z]+)(\d+)", base)
  if not match:
     return None
  letters, digits = match.groups()
  return f"{letters.lower()}_1_{int(digits)}"

#Import images
def import_images_from_folder(folder_id):
  query = f"'{folder_id}' in parents and mimeType contains 'image/'"
  files = drive_service.files().list(q=query, fields="files(id, name)").execute().get("files", [])

  for file in files:
     file_name = file['name']
     location_id = standardize_location_id(file_name)

     if not location_id:
            print(f"Skipping invalid image name: {file_name}")
            continue

        url = f"https://drive.google.com/uc?export=view&id={file['id']}"
        cursor.execute("""
            INSERT OR IGNORE INTO images (location_id, file_name, date, url)
            VALUES (?, ?, ?, ?)
        """, (location_id, file_name, DEFAULT_DATE, url))
        print(f"Inserted image: {file_name} → {location_id}")

# Sheets to tables
def import_from_sheet():
    sheet = sheet_client.open(SHEET_NAME).sheet1
    records = sheet.get_all_records()

    for row in records:
        location_id = standardize_location_id(row['Photo ID'])
        if not location_id:
            print(f"Skipping invalid Photo ID: {row['Photo ID']}")
            continue

        building_name = row['Building Name'].lower().replace(" ", "_")
        floor_number = 0 if 'ground' in row['Floor'].lower() else 1
        side = row['Side of Building'].lower()

        try:
            dt = datetime.strptime(row['Time (24h)'], "%m/%d/%Y %H:%M:%S")
            date_str = dt.strftime("%Y-%m-%d")
            hour = dt.hour
        except Exception as e:
            print(f"Skipping row with bad time format: {row['Time (24h)']}")
            continue

        # Insert into static
        cursor.execute("""
            INSERT OR IGNORE INTO static (location_id, building_name, floor_number,
side_of_building)
            VALUES (?, ?, ?, ?)
        """, (location_id, building_name, floor_number, side))
        # Get URL
        cursor.execute("SELECT url FROM images WHERE location_id = ?", (location_id,))
        result = cursor.fetchone()
        url = result[0] if result else None

        #Insert into environment
        cursor.execute("""
            INSERT INTO environment_logs (
                location_id, outside_temp, min_temp, max_temp,
                time_taken_hours, windows_opened, date, url
            ) VALUES (?, ?, ?, ?, ?, ?, ?, ?)
        """, (
            location_id,
            float(row['Outdoor Temp']),
            float(row['Min Temp (C)']),
            float(row['Max Temp (C)']),
            hour,
            'N' if row['Framed Properly?'].strip().lower() == 'yes' else 'Y',
            date_str,
            url
        ))

        print(f"Inserted environment log for {location_id}")

#Run everything
for label, folder_id in DRIVE_FOLDER_IDS.items():
    print(f"Importing images from: {label}")
    import_images_from_folder(folder_id)

print("Importing static/environment_logs from Google Sheets...")
import_from_sheet()

conn.commit()
conn.close()
print("All data imported successfully.")
```