

Thermal Vision

An Android Thermal Image Processing App

A Project-II Report

Submitted in partial fulfillment of requirement of the

Degree of

**BACHELOR OF TECHNOLOGY in COMPUTER
SCIENCE & ENGINEERING**

BY

Lakshmi Srikumar

EN21CS301416

Under the Guidance of

Prof Divya Kumawat

Mr. Rajiv Jain



Department of Computer Science & Engineering

Faculty of Engineering

MEDICAPS UNIVERSITY, INDORE- 453331

May 2025

Acknowledgements

I would like to express my deepest gratitude to Honorable Chancellor, **Shri R C Mittal**, who has provided me with every facility to successfully carry out this project, and my profound indebtedness to **Prof. (Dr.) D. K. Patnaik**, Vice Chancellor, Medicaps University, whose unfailing support and enthusiasm has always boosted up my morale. I also thank **Prof. (Dr.) Pramod S. Nair**, Dean, Faculty of Engineering, Medicaps University, for giving me a chance to work on this project. I would also like to thank my Head of the Department **Dr. Ratnesh Litoriya** for his continuous encouragement for the betterment of the project.

I express my heartfelt gratitude to my External Guide, **Shri. Rajiv Jain, Head of Software Development, LCID, RRCAT** as well as to my Internal Guide, **Smt. Divya Kumawat**, Professor, Department of Computer Science Engineering, Medi-Caps University, without whose continuous help and support, this project would ever have reached to the completion.

I would also like to thank **Shri PP Deshpande, Associated Director, Laser Group and Head, Laser Control and Instrumentation Division** as well as **Shri. Sandeep Talwar, Shri Sarthak Gupta** of RRCAT who extended their kind support and help towards the completion of this project.

It is their help and support, due to which we became able to complete the design and technical report.

Lakshmi Srikumar

B.Tech. IV Year

Department of Computer Science & Engineering

Faculty of Engineering

Medicaps University, Indore

Abstract

Thermal imaging is a powerful, non-contact technique widely used in applications such as industrial inspection, photovoltaic (PV) module monitoring, electrical diagnostics, and preventive maintenance. These thermal images, however, often require further processing to extract meaningful temperature data and accurately identify critical regions of interest such as hotspots. This project focuses on the development of a mobile application designed to process thermal images and detect temperature variations using advanced digital image processing techniques.

The application is developed for the Android platform using Android Studio with Kotlin and integrates the scikit-image library for image processing operations. The system enables users to upload or capture thermal images using their Android device, which are then processed through a defined pipeline. This pipeline includes normalization of image intensity values, Gaussian and median blurring to reduce noise, and Contrast Limited Adaptive Histogram Equalization (CLAHE) to enhance local contrast. These preprocessing steps significantly improve the visual quality of the thermal data and support more accurate downstream analysis.

Following enhancement, the application performs hotspot detection using color-based segmentation in the HSV color space. It then maps pixel intensity values to real-world temperatures using a configurable conversion function. The processed image is displayed in a user-friendly interface that allows real-time interaction with parameters such as minimum and maximum temperature thresholds, contrast levels, and kernel sizes for filtering operations.

The application also features chronological tracking of processed images, supporting comparison over time, and lays the groundwork for future capabilities such as real-time data acquisition from connected thermal cameras, cloud-based analytics, and machine learning-driven anomaly classification. By combining mobile accessibility with powerful image processing features, this project provides a cost-effective, portable, and scalable solution for on-site thermal diagnostics and monitoring, particularly in environments where high-end equipment may be impractical or cost-prohibitive.

Table of Contents

	Page No.
Report Approval	i
Declaration	ii
Certificate	iii
Acknowledgement	iv
Abstract	v
Table of Contents	vi
List of figures	vii
Abbreviations	viii
Notations & Symbols	ix
 Chapter 1	
Introduction	
1.1 Introduction	1
1.2 Literature Review	2
1.3 Objectives	4
 Chapter 2	
Implementation	
2.1 Introduction to Languages, IDE's, Tools and Technologies	6
2.2 Requirement Specification	9
2.3 System Design	11
2.4 Performance Profiling	16
2.5 End User Instruction	21
 Chapter 3	
Results & Discussion	
3.1 Results	23
3.2 Future Scope	25
3.3 Conclusion	26
 Bibliography	27
Appendix	28


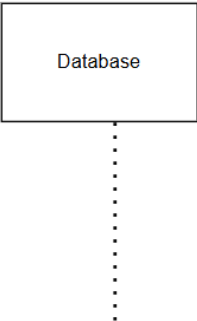
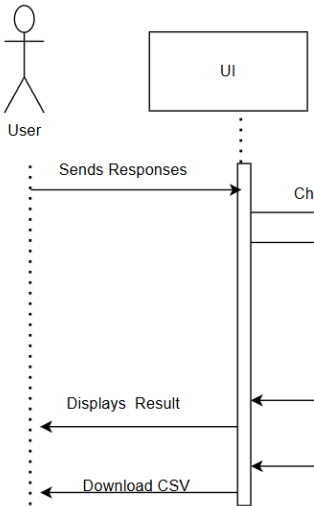
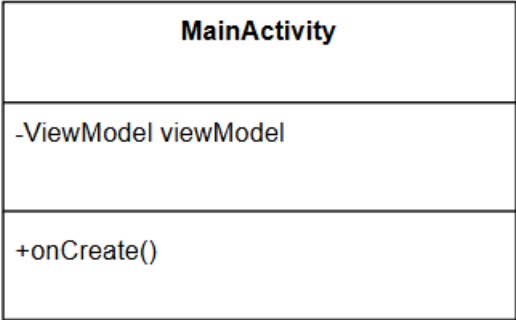

List of Figures

Figure	Title	Page No
Fig 1	Spectral sensitivity functions classified into the two categories of (a) iOS phone cameras, and (b) Android phone cameras.	3
Fig 2	Estimated RGB spectral response functions of the built-in cameras of high-end and low-end smartphones. The recovered RGB spectral response functions of (a) Samsung Galaxy Note 8 (b) Galaxy S9+ and (c) Galaxy A21.	4
Fig 3	Model – View – View Model Architecture	9
Fig 4	Sequence Diagram	11
Fig 5	Class Diagram	13
Fig 6	Profiling of the start up time of the app	17
Fig 7	Instance of Janky UI frame during runtime	18
Fig 8	Display profile of the app during its runtime	18
Fig 9	The battery chart indicating the usage of battery during the app's lifecycle	19
Fig 10	CPU performance profiling during the app lifecycle	20
Fig 11	Physical Memory of the app in the Memory profiler	20
Fig 12	Memory profiling of the app	21
Fig 13	User Interface of the Project (a) Landing Screen (b) Loading Screen (c) Analysis Page .	24
Fig 14	Screenshot of the CSV containing the multi-touch analysis of the processed image	24
Fig 15	The Project Structure	28

Abbreviations

Abbreviation	Full Form
IR	Infrared Radiation
DIP	Digital Image Processing
PV	Photovoltaic
CLAHE	Contrast Limited Adaptive Histogram Equalization
CPU	Central Processing Unit
GPU	Graphics Processing Unit
ISO	International Standards Organization
RGB	Red , Green , Blue
JPEG	Joint Photographic Experts Group
API	Application Programming Interface
IDE	Integrated Development Environment
SDK	Software Development Kit
APK	Android Application Package
AVD	Android Virtual Device
HSV	Hue , Saturation , Value
MVVM	Model , View , View- Model
UI	User Interface
UX	User Experience
UML	Unified Modeling Language
TIFF	Tagged Image File Format
PNG	Portable Network Graphics
CSV	Comma-Separated Values
RAM	Random Access Memory
GB	Gigabyte
SQL	Structured Query Language
No SQL	Non Structured Query Language

Notations & Symbol

Symbol / Notation	Meaning
	Actor - represents a type of role where it interacts with the system and its objects.
	Lifeline - element which depicts an individual participant in a sequence diagram. Here X is the object or instance name Class 1 is the class name.
	Messages - Communication between objects is depicted using messages. The messages appear in a sequential order on the lifeline.
	<p>Class Name: The name of the class is typically written in the top compartment of the class box and is centered and bold.</p> <p>Attributes: also known as properties or fields, represent the data members of the class. They are listed in the second compartment of the class.</p> <p>Methods: also known as functions or operations, represent the behavior or functionality of the class. They are listed in the third compartment of the class box.</p> <p>Visibility Notation: It indicates the access level of attributes and methods. Common visibility notations include:</p> <ul style="list-style-type: none"> + for public (visible to all classes) - for private (visible only within the class) # for protected (visible to subclasses)
	Directed association - represents a relationship between two classes where the association has a direction, indicating that one class is associated with another in a specific way.

Chapter –1 Introduction

1.1 Introduction

Infrared (IR) radiation is a type of electromagnetic radiation with wavelengths longer than visible light, typically ranging from 700 nanometers (nm) to 1 millimeter (mm). All objects with a temperature above absolute zero emit infrared radiation as a function of their thermal energy. The intensity and wavelength of the emitted radiation vary with the object's surface temperature, making infrared detection a powerful tool for temperature measurement and thermal pattern visualization.

Thermal cameras, also known as infrared cameras, are designed to detect this invisible IR radiation and convert it into an image that visually represents temperature variations across a scene. The raw data captured by thermal sensors is usually in the form of high-bit-depth grayscale images where each pixel corresponds to an infrared intensity value. However, these images are often noisy, low in contrast, and not readily interpretable by the human eye. This is where **Digital Image Processing (DIP)** plays a crucial role.

Digital Image Processing refers to the use of computational algorithms to manipulate and analyze digital images to extract useful information, enhance visual clarity, and enable automated interpretation. In the context of thermal cameras, DIP techniques are essential for improving image quality, removing noise, enhancing contrast, isolating regions of interest (such as hotspots), and converting pixel intensity values into temperature readings. Common DIP methods used in thermal image processing include normalization, filtering, histogram equalization, segmentation, edge detection, and colormap application. This project leverages the capabilities of DIP to build a thermal analysis system on a mobile platform. Specifically, the objective is to develop an Android application that processes thermal images for temperature detection.

The motivation behind this project is to make thermal analysis more accessible by utilizing the capabilities of mobile devices. Traditional thermal analysis systems require expensive hardware and specialized software, limiting their use to high-budget industrial or research environments. In contrast, this Android-based application provides a lightweight and intuitive alternative that can be used in a variety of domains such as photovoltaic module inspection, electrical diagnostics, machinery health monitoring, and more.

In essence, this project bridges the gap between professional thermal imaging systems and consumer-grade mobile platforms by combining the physics of infrared sensing with the power of digital image processing, delivering an efficient, accurate, and user-friendly tool for thermal diagnostics.

1.2 Literature Review

Early studies established that infrared (IR) thermography is a powerful non-destructive method for finding defects in photovoltaic (PV) systems. For example, Tsanakas and Botsaris ^[1] demonstrated an IR-based tool that analyzes PV module thermograms using image histograms and line profiles. They showed that distinct features in these thermal-image histograms and linear temperature profiles reliably indicate the presence and severity of hot-spot defects on PV modules.

In parallel with advances in PV inspection, the 2010s saw rapid progress in mobile-phone thermal imaging hardware and software. Around 2014–2015, consumer-grade IR sensors (e.g. FLIR One, Seek Thermal) began to appear as smartphone attachments, making thermal cameras affordable and portable. Building on this trend, Lee *et al.* ^[2] proposed a complete **Mobile Thermal Imaging System (MTIS)** using a smartphone. Their MTIS consisted of a small thermal-infrared module (TIM) attached to the phone and a custom Android app (“IRAPP”) for image processing. Importantly, Lee *et al.* optimized the software to fully use the phone’s CPU/GPU: they implemented non-uniformity correction and scene-change detection algorithms on the phone itself, rather than merely using the phone as a display terminal. In tests, the MTIS achieved a practical detection range of about 29 m for temperature targets. They even discussed novel applications – for instance, using the smartphone imager to monitor infant body temperature for SIDS prevention. This work demonstrated that a mobile phone with an attached IR sensor can form a *self-contained* thermal imager, leveraging wireless connectivity and powerful apps to extend IR diagnostics into everyday civilian use.

As smartphone IR imaging took off, researchers also began to carefully characterize and calibrate phone cameras (both IR and visible) for scientific use. Burggraaf *et al.* ^[3] developed a standardized calibration methodology (called SPECTACLE) for consumer cameras, including smartphones. They found that while RAW sensor outputs are typically highly linear, most camera pipelines (e.g. JPEG) are not linear and exhibit complex ISO/gain behavior. Crucially, their experiments revealed **large pixel-level and device-level variability**: for example, some camera sensors showed over 400% inter-pixel gain variation (worst pixels were 4× brighter than others) and flat-field correction factors that varied by up to 2.79 across an image. They also found significant differences in spectral response curves between camera models – i.e. no two smartphone cameras responded to colors or infrared wavelengths in the same way. These results imply that per-device calibration is essential when using phone cameras as quantitative sensors. In other words, to use a smartphone for serious imaging (thermal or otherwise), one must correct for its sensor non-uniformity and unique color response – a fact underscored by these studies of consumer-camera calibration.

Building on the need for calibration, Shoji Tominaga ^[4] focused on **spectral sensitivity functions** of smartphone cameras. Spectral sensitivity functions describe how much response (in the R, G, B channels) a camera has at each wavelength. Since manufacturers do not publish these data, Tominaga developed methods to **measure and estimate** them. He constructed an imaging system with monochromatic light to directly measure each phone camera's RGB sensitivity curve, and devised an indirect estimation method using color charts and principal-component analysis. Using these tools, he measured a variety of mobile-phone cameras and built a **database of spectral sensitivity functions**. His work showed that each smartphone has a distinctive RGB response profile, but these profiles can be characterized accurately either by direct calibration or by smart estimation from known color targets. In practical terms, this means that apps can be programmed with the phone's spectral sensitivity and thus translate raw RGB values into actual color or temperature readings more accurately.

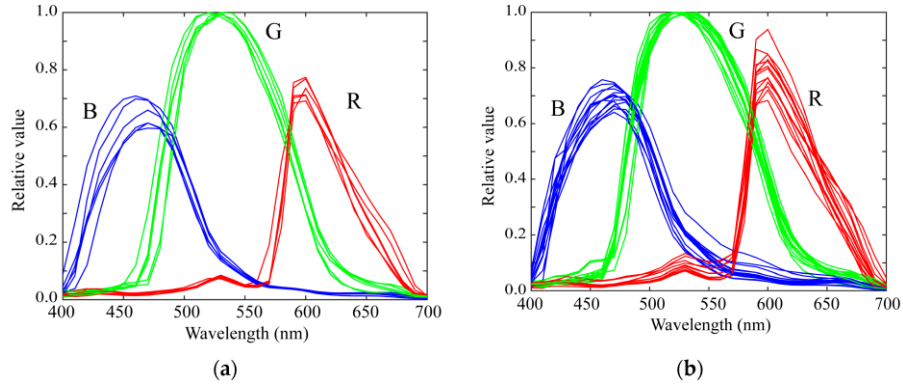


Fig 1: Spectral sensitivity functions classified into the two categories of (a) iOS phone cameras, and (b) Android phone cameras.

Another recent line of work used advanced algorithms to recover smartphone camera curves from very limited data. Lu *et al.* ^[5] applied a compressive sensing approach using only 12 color patches to estimate RGB response functions. They validated their method on high-end and low-end phones: for Samsung Galaxy Note8 and S9+, the estimated curves could reproduce measured colors with very low error (RMS relative error $\sim 7\text{--}8\%$), whereas for a budget Galaxy A21 the error was higher ($\sim 10\%$) due to greater sensor noise. This study reinforces the finding that high-end phones yield more reliable data, but also that even with minimal inputs one can extract each camera's response. Such calibration databases and algorithms mean that a mobile app can account for the phone's own limitations when interpreting its images.

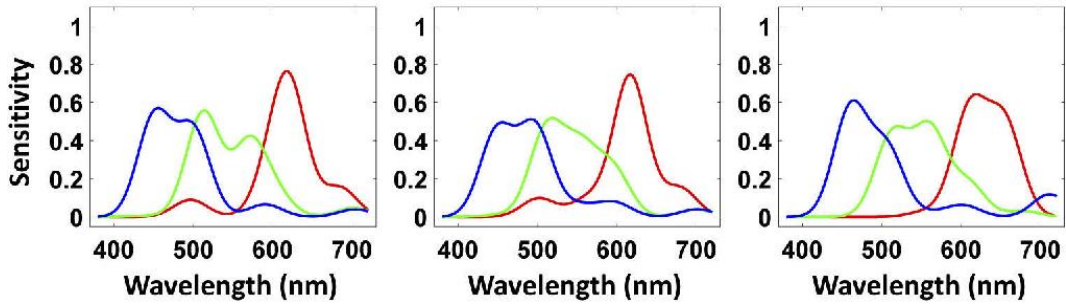


Fig 2. Estimated RGB spectral response functions of the built-in cameras of high-end and low-end smartphones. The recovered RGB spectral response functions of (a) Samsung Galaxy Note 8, (b) Galaxy S9+& (c) Galaxy A21.

1.3 Objective

The project aims to develop a comprehensive, Android-based thermal image analysis platform that bridges the gap between advanced thermal diagnostics and mobile accessibility. By leveraging digital image processing techniques and modern Android frameworks, the application seeks to empower professionals across industries to perform precise, on-device thermal inspections with minimal reliance on specialized hardware. Below is a detailed breakdown of the project's technical, functional, and strategic objectives:

1.3.1 Core Technical Objectives

1.3.1.1. Thermal Image Acquisition and Preprocessing

Objective: Enable seamless acquisition of thermal images through multiple input sources while ensuring compatibility with industry-standard formats.

Sub-Goals:

- Support 16-bit TIFF file decoding to retain high-resolution thermal data.
- Integrate Android Camera2 API for capturing images using smartphone cameras.
- Implement gallery import functionality for pre-recorded thermal images.

1.3.1.2. Image Enhancement and Segmentation

Objective: Optimize thermal images for accurate temperature mapping through advanced preprocessing techniques.

Sub-Goals:

- Apply noise reduction algorithms to enhance raw image quality.
- Develop algorithms to improve contrast in low-dynamic-range thermal images.
- Implement adaptive thresholding to isolate regions of interest (ROIs) based on pixel intensity.

1.3.1.3. Temperature Mapping and Hotspot Detection

Objective: Convert pixel data into actionable temperature metrics

Sub-Goals:

- Design a calibration module to map pixel intensity values to temperature gradients
- Develop algorithms to detect hotspots by analyzing local maxima in temperature distributions.
- Calculate statistical metrics (min, max, average) for user-selected regions.
- Enable multi-point analysis to compare temperature variations across different areas.

1.3.2. Functional Objectives

1.3.2.1 User-Centric Interface Design

Objective: Deliver an intuitive, responsive interface tailored for field technicians and engineers.

Sub-Goals:

- Implement touch-based temperature probing, allowing users to tap any image region to retrieve real-time readings.
- Design dynamic heatmap visualization using false-color palettes for intuitive interpretation.

1.3.2.2 Data Management and Reporting

Objective: Streamline data storage and export workflows for seamless integration with desktop tools.

Sub-Goals:

- Store metadata (timestamp, min/max temperatures) in lightweight CSV files.
- Generate export-ready reports with temperature statistics and annotated hotspot coordinates.

1.3.2.3 Real-Time Performance Optimization

Objective: Ensure efficient processing on mid-range Android devices.

Sub-Goals:

- Optimize algorithms for latency below 500ms on 1080p images (e.g., using multithreading with Kotlin Coroutines).
- Minimize memory footprint through bitmap compression and lazy loading.
- Test compatibility across Android API levels 33+ (Android 13 and above).

Chapter –2 Implementation

2.1 Introduction to Languages, IDE's, Tools, and Technologies

2.1.1 Languages Used:

- **Kotlin:**

The Android app is written primarily in **Kotlin**, with image-processing prototypes developed in **Python**. Kotlin has been the preferred language for Android development since 2019, and over half of professional Android developers now use Kotlin as their main language. Its concise, expressive syntax and strong safety features (null-safety, reduced boilerplate) improve developer productivity and reduce runtime crashes. Importantly, Kotlin is fully interoperable with existing Java and Android APIs, easing integration into the Android ecosystem.

- **Python:**

Python, by contrast, was chosen for prototyping the image-processing algorithms because of its simplicity, rich libraries, and rapid development cycle. Python's extensive computer-vision libraries (OpenCV, scikit-image, etc.) and clear syntax enable quick prototyping and experimentation with complex image algorithms. In practice, Python code was used to test and refine the noise-reduction, contrast, and contour-detection routines before porting them into Kotlin on Android.

2.1.2 IDE Used:

- **Android Studio:**

Development was conducted in **Android Studio**, the official Android IDE. Android Studio is built on JetBrains' IntelliJ IDEA platform and is specifically optimized for Android. It integrates the Android SDK, build tools (Gradle), an emulator, visual layout editors, and debugging tools into one environment. Using Android Studio ensured seamless support for Kotlin (including Android KTX and Jetpack libraries) and provided features like code completion, real-time profilers, and UI preview. The built-in project templates and debugging aids (e.g. Logcat, Layout Inspector) streamlined the development of the thermal-image app. In short, Android Studio's tight integration with the Android toolchain made it the natural choice for building, testing, and deploying the mobile application.

2.1.3 Tools and Libraries:

- **Scikit-Image:**

It was selected as the primary library for performing thermal image processing tasks instead of the more commonly used OpenCV. Scikit-image, part of the broader SciPy ecosystem, is a lightweight, efficient, and pure Python library specifically designed for image processing. It provides a wide range of image manipulation functions, which are essential for preparing thermal images for analysis. One of the main reasons for choosing scikit-image over OpenCV was its **smaller file size and lighter runtime footprint**, making it highly suitable for mobile-based applications where app size and resource usage must be optimized.

- **Gradle:**

The project uses **Gradle** as its build automation and dependency-management system. Gradle is the standard Android build tool, designed for multi-language projects. It handles compiling the Kotlin code, processing resources, and packaging the APK. Gradle's declarative configuration allows easy inclusion of libraries and simplifies managing external dependencies. Using Gradle ensured efficient, repeatable builds and easy configuration of SDK versions and library settings specific to this project.

- **Android Virtual Device (AVD) Emulator:**

For testing, an **AVD Emulator** was employed to simulate Android devices. The Android Emulator (bundled with Android Studio) lets developers test apps across different device models, screen sizes, and API levels without needing physical hardware. It provides high-fidelity device behavior (simulating camera, sensors, network, etc.) and rapid deploy-run cycles. In this project, the AVD allowed rapid iteration and debugging of the thermal app (for example, testing the UI responsiveness and image-capture functions). In short, the emulator's flexibility and speed made development and testing more efficient.

- **Chaquopy:**

To enable seamless integration of Python-based image processing within the Android environment, this project utilized Chaquopy (the Python SDK for Android). Chaquopy allows developers to run native Python code alongside Kotlin or Java in Android applications without requiring external servers or complicated bridges. Moreover, Chaquopy's integration with Gradle and Android Studio provided a smooth development experience, allowing Python modules to be managed just like regular Android dependencies. This greatly simplified the architecture of the application, enabling powerful, real-time thermal image analysis while maintaining the lightweight, offline-first design essential for mobile applications.

2.1.4 Technologies and Concepts:

Several key image-processing and mobile-app concepts were integral to the implementation:

- **Digital Image Processing:**

The core thermal analysis uses classical DIP techniques to prepare and enhance the images. Raw thermal data (often 16-bit radiometric images) must first be normalized or contrast-stretched into an 8-bit range, bringing the pixel intensities into a usable range. Noise reduction is then applied via **Gaussian blurring** (a low-pass filter that smooths image regions) and **median filtering** (effective at removing salt-and-pepper noise while preserving edges). After denoising, contrast is locally enhanced using **CLAHE** (Contrast Limited Adaptive Histogram Equalization) which adaptively boosts contrast in local regions without over-saturating the image. These preprocessing steps (normalization, blurring, and adaptive contrast) improve image clarity and prepare the data for robust temperature detection.

- **HSV Color Segmentation:**

To identify hot regions in the thermal image, the processed image is mapped through a false-color JET colormap and then converted to the HSV color space. In HSV space, color segmentation is more intuitive: hue and saturation capture the color characteristics independent of brightness. By thresholding the hue channel for red tones (typical of the “hot” end of the colormap), red regions can be isolated reliably. This HSV-based color segmentation allows the app to generate a binary mask of potential hotspots (regions of high temperature) even under varying lighting or image conditions. Although simple, this color-thresholding approach is effective for detecting the highest-temperature regions in the thermal image.

- **Temperature Mapping:**

Converting pixel values to actual temperature readings is crucial for the application’s purpose. Radiometric thermal cameras encode temperature data in their pixels, which must be converted via known calibration factors. For example, FLIR cameras ^[6] use a linear scale factor to map stored pixel values to physical temperature. This mapping (from gray-level or color-coded value to temperature) leverages the camera’s radiometry so that the app can display accurate temperature readings for any selected point or region.

- **Mobile Application Architecture:**

The Android app’s architecture was designed for responsive, real-time user interaction. Common patterns like Model–View–View Model (MVVM) ^[7] were used to separate UI code from business logic. For instance, image-processing operations and temperature computations run on background threads or coroutines to avoid blocking the UI, while Live Data or observable properties

update the view in real time. The UI is responsive and adaptive, adjusting to different screen sizes/orientations (responsive design) and providing user controls (sliders, buttons) to tweak processing parameters (e.g. blur kernel size or color thresholds). This structured mobile architecture (leveraging Android’s architecture components) ensures that image frames can be processed and displayed continuously without lag, and that the user can interactively adjust settings without destabilizing the application.

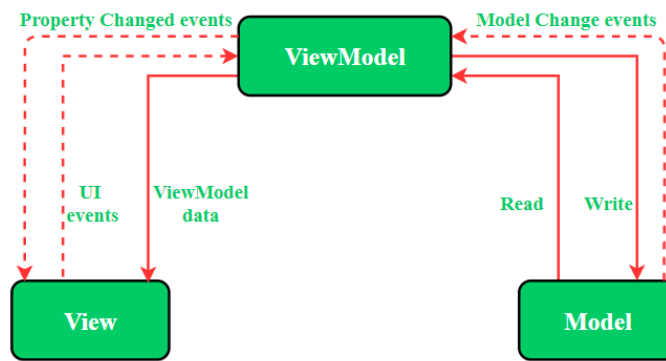


Fig 3. Model – View – View Model Architecture

2.2 Requirement Specification

2.2.1 Functional Requirements

The thermal image processing application must allow users to upload or capture thermal images in formats like .jpg, .png, or .tiff. The user interface should provide intuitive controls to select an image, view it immediately, and move forward with analysis. It should also allow users to dynamically adjust essential parameters before processing the image.

The core functionality requires the system to perform real-time image processing operations once the user submits the selected image. It must apply normalization, smoothing filters, contrast enhancement and color-based segmentation. Additionally, the application should accurately detect and annotate hotspots, convert pixel values to temperature values, and present the processed thermal image with clear overlays to the user without any notable delay or crash.

Another crucial requirement is **user interactivity**: the app must allow users to touch any point on the processed thermal image and immediately display the corresponding temperature at that coordinate. Furthermore, the app should provide options to download the extracted data into a structured CSV file, recording temperature readings, coordinates, and time, ensuring users have access to results for further analysis or reporting.

2.2.2 Software Requirements

The project development environment primarily requires Android Studio as the Integrated Development Environment (IDE) to build, test, and deploy the application. Android Studio provides complete support for Kotlin development, Gradle build management, and emulators for testing the app on various screen sizes and API versions. Chaquopy must also be installed as a plugin for Android Studio to enable the integration of Python code inside the Android application.

At the programming level, Kotlin is used for handling the Android front-end (UI/UX, navigation, event handling), while Python (via Chaquopy) is used for executing heavy image processing tasks like normalization, Gaussian/Median blur, CLAHE, and pixel intensity mapping using scikit-image libraries. Thus, the software stack must support both Kotlin and embedded Python execution.

Furthermore, the mobile device where the app is installed should have access to basic libraries and APIs provided by the Android SDK, including permission handling for file storage access, touch screen event handling, and external storage management for exporting CSV files. Additionally, the Android device needs internet or local storage access to import images for testing, although the app is designed to work offline once installed.

2.2.3 Hardware Requirements

The hardware for development and testing requires a computer system (laptop or desktop) with at least:

- A multi-core processor (Intel i5 / Ryzen 5 equivalent or better),
- Minimum 8 GB RAM (preferably 16 GB for smooth Android Studio operation),
- At least 20 GB free disk space for Android SDKs, emulators, and project files. A dedicated graphics card can further accelerate emulator operations, but it is optional.

On the mobile device side, the Android smartphone must be capable of running moderate computational tasks. A device with:

- Minimum 3 GB RAM,
- A mid-range or better processor (Snapdragon 600 series or equivalent),
- And Android OS version 8.0 (Oreo) or higher, is recommended to ensure smooth functioning without delays, especially during the real-time processing steps.

Sufficient storage (at least a few hundred MB free) is also needed to store processed images and exported CSV files locally.

2.3 System Design

System design is the process of defining the architecture, components, modules, interfaces, and data flow of a software system to meet specific functional and non-functional requirements. It acts as a blueprint for developers, outlining how various parts of the application will interact, how data will move through the system, and how responsibilities will be distributed across components.

UML (Unified Modeling Language) is the standard method used to visually represent the structure and behavior of a system during design. It provides a set of diagrams that describe different aspects of the system.

A **Sequence Diagram** is a type of behavioral UML diagram that illustrates how objects or components in a system interact with each other over time. It focuses on the **order and timing of messages** exchanged between system elements (such as classes, users, or modules) during a specific scenario or process.

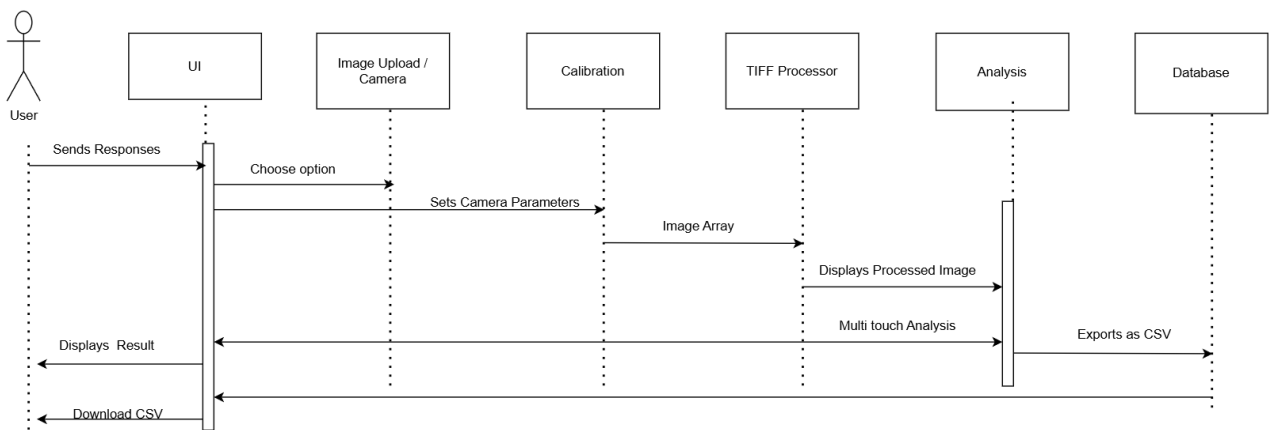


Fig 4. Sequence Diagram

1. Image Capture / Upload:

The user initiates the process by either capturing a thermal image using a camera module (external or phone-based) or selecting an existing thermal image from the device storage.

The thermal image is typically a raw 16-bit grayscale TIFF format, where each pixel intensity is directly proportional to the detected temperature.

2. Calibration:

User-defined parameters such as minimum and maximum temperature thresholds are set for the upcoming image processing steps. This configuration guides how the thermal data will be interpreted. Once calibration is complete, the system forwards the raw 16-bit image data to the TIFF Processor, which is responsible for extracting and preparing the pixel values for further processing.

3. TIFF Processor:

The raw pixel data is passed into the Python environment through Chaquopy.

Here, image preprocessing operations are performed:

- Normalization: The 16-bit raw pixel data is scaled down to an 8-bit (0–255) range standardization.
- Optional Smoothing: Additional operations like Gaussian Blur or Median Blur can be applied to reduce noise and smoothen the image.
- CLAHE: Applied to locally enhance the contrast of the image, making temperature differences more visible.

The output of this Python module is a processed temperature array, which is a cleaned, enhanced version of the thermal data ready for analysis.

4. Analysis:

Back in Kotlin and via Python, the following analysis steps are performed:

- Color Mapping: The normalized image is optionally converted into a false-color JET colormap to visually represent temperature levels.
- HSV Conversion and Thresholding: The processed image is converted into HSV color space, and thresholding is performed to detect regions corresponding to high temperatures (hotspots).
- Temperature Mapping: The average pixel value inside each detected hotspot is calculated and mapped to actual temperature values using a calibrated linear formula.

5. Data Export:

After successful analysis, the temperature data (location, calculated temperature) is saved in a CSV file.

Instead of a traditional SQL or NoSQL database, a lightweight CSV export is used for simplicity:

Each record represents a detected hotspot with attributes such as Image name, Temperature value, X, Y coordinates. This export allows easy offline analysis, sharing, or further integration with larger systems if needed.

6. User Interface (UI) Updates:

The app listens for touch events on the displayed image.

When the user touches a point:

- The (x, y) pixel coordinates of the touch are captured.
- The corresponding pixel intensity value from the processed temperature array is retrieved.
- The intensity value is converted into the corresponding temperature using the same calibration formula.

The app then displays the exact temperature at the touched location on the screen as a label below the list of temperature result.

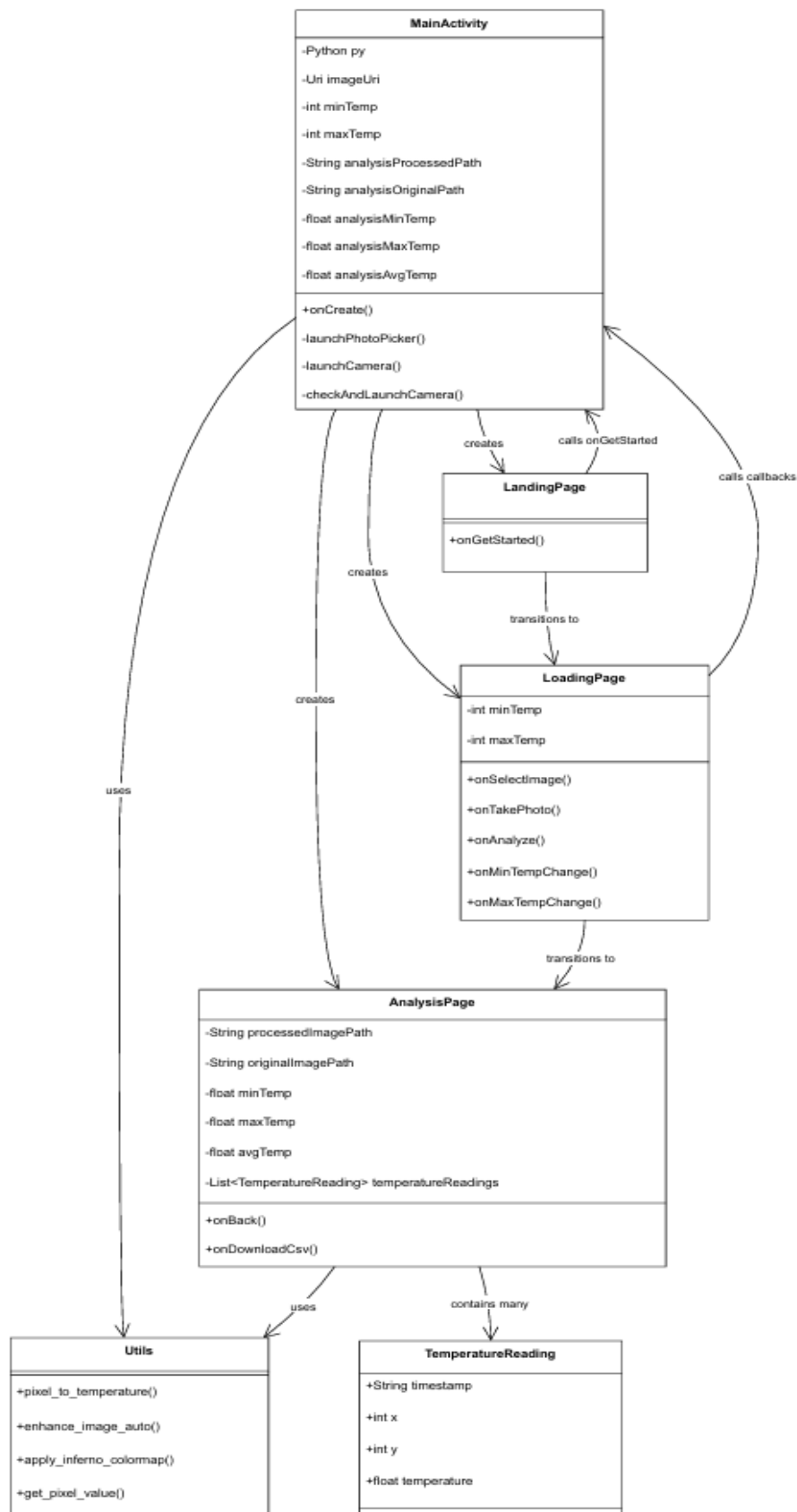


Fig 5. Class Diagram

A Class Diagram is a type of UML diagram that visually represents the static structure of a software system. It shows the system's classes, their attributes, methods (operations), and the relationships between them, such as inheritance, association, and composition.

Each class in the diagram is typically drawn as a rectangle divided into three compartments:

1. **Top** – Class name
2. **Middle** – Attributes (data members or properties)
3. **Bottom** – Methods (functions or operations the class can perform)

The following are the classes and the roles that they perform:

1. Main Activity:

- Central controller class of the app.
- Manages core state variables such as image Uri, minTemp, maxTemp, and processed image paths.
- Contains lifecycle methods like onCreate() and methods to launch the photo picker or camera.
- Responsible for navigating between pages and initiating processing workflows.

2. Landing Page

- The starting screen of the app.
- Provides the “Get Started” UI and transitions the user to the next page.
- Triggered by onStart().

3. Loading Page

UI page where users configure temperature ranges and choose how to load the image.

Methods include:

- onSelectImage(): Opens image picker.
- onTakePhoto(): Launches camera.
- onAnalyze(): Initiates analysis.
- onMinTempChange() & onMaxTempChange(): Respond to slider interactions.

Transitions to Analysis Page after processing is triggered.

4. Analysis Page

- Displays the results of image processing.
- Stores min/max/avg temperature and image paths.
- Holds a list of Temperature Reading objects to display temperature values per coordinate.

Methods:

- onCallback(): Handles post-processing UI updates.
- onDownloadCsv(): Triggers CSV export of the results.

5. Utils.py

A utility/helper python class used by multiple pages to perform:

- pixel_to_temperature(): Converts pixel intensity to temperature.
- enhance_image (): Applies image enhancement.
- apply_colormap(): Applies color mapping to thermal data.
- get_pixel_value(): Retrieves pixel value at a touch coordinate.

6. Temperature Reading

Model class for individual temperature records.

Contains fields:

- Timestamp, x, y, temperature.
- Used to populate the list in Analysis Page and export data to CSV.

Interaction Between Methods

1. Main Activity creates and displays the Landing Page, which waits for onStart() to be triggered.
2. When the user starts the app:
 - Landing Page transitions to LoadingPage.
3. In Loading Page:
 - Users set temperature thresholds (onMinTempChange, onMaxTempChange).
 - Select or capture an image using onSelectImage() or onTakePhoto().
 - Trigger analysis with onAnalyze().
4. Utils methods are called by Main Activity or LoadingPage to:
 - Process the image,
 - Apply enhancements,
 - Extract temperature data from pixel values.
5. Once processing is done, the app transitions to Analysis Page, passing all computed temperature values and image paths.
6. AnalysisPage:
 - Displays processed results.
 - Shows a list of TemperatureReading objects.
 - Allows CSV export through onDownloadCsv().
7. Temperature Reading holds and formats the temperature at selected points on the image.

2.4 Performance Profiling

Performance is a critical aspect of any Android application. A slow, unresponsive, or battery-draining app leads to poor user experience and negative reviews. As modern Android applications become more complex, ensuring smooth performance is essential for user retention.

Android Studio provides a powerful set of tools for profiling and debugging applications, helping developers identify performance bottlenecks, optimize CPU and memory usage, and reduce unnecessary power consumption.

What is Profiling?

Profiling in Android development involves analyzing the app's runtime behavior, including CPU usage, memory allocation, UI responsiveness, and network performance. It helps developers diagnose issues and optimize their apps for better performance.

Why is Profiling Important?

- **Detects Memory Leaks:** Prevents excessive memory consumption leading to Out of Memory Errors.
- **Optimizes CPU Usage:** Identifies unnecessary processing that can slow down the app.
- **Improves UI Responsiveness:** Detects janky frames and laggy interactions.
- **Reduces Battery Consumption:** Helps find areas where excessive power is being used.

Profiling Tools in Android Studio

Android Studio offers a built-in **Profiler** tool that provides real-time monitoring of CPU, memory, and network usage.

Key Profiling Tools:

1. **CPU Profiler:** Analyzes how the app utilizes the CPU and detects slow methods or inefficient algorithms.
2. **Memory Profiler:** Monitors memory usage and detects memory leaks.
3. **Network Profiler:** Shows network requests and response times.
4. **Energy Profiler:** Helps identify power-consuming components of an app.

2.4.1 Analyzing Performance bottlenecks

2.4.1.1 App Takes a Long Time to Start

One of the most noticeable performance issues in mobile apps is a **long startup time** — the delay between tapping the app icon and when it becomes usable. From a user's perspective, this creates frustration and the impression that the app is poorly designed or unstable.

Ideal Startup Time:

According to Android performance guidelines:

- **Cold Start (first launch after reboot or system clear):**
Should ideally complete within **5 seconds**.
- **Warm Start (app already cached in memory):**
Should complete within **1–2 seconds**.

If an app takes more than **5–6 seconds** to load on a cold start, it risks being perceived as unresponsive — especially if no visual feedback (e.g., a splash screen or loading animation) is provided.

Observation: The app successfully achieves startup times (cold start) under **2 seconds**.

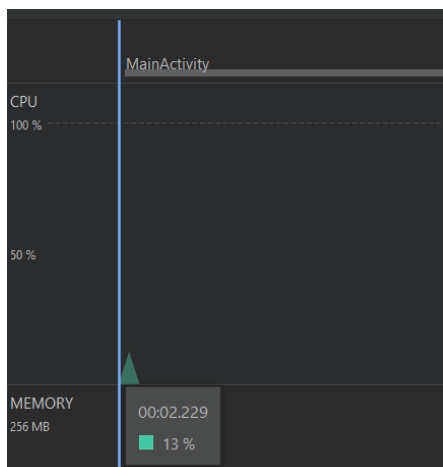


Fig 6 : Profiling of the start up time of the app

2.4.1.2 Janky/Slow UI

Android renders UI by generating a frame from your app and displaying it on the screen. If your app suffers from slow UI rendering, then the system is forced to skip frames. When this happens, the user perceives a recurring flicker on their screen, which is referred to as jank.

When jank occurs, it's usually because of some deceleration or blocking async call on the UI thread (in most apps, it's the main thread). You can use system traces to identify where the problem is.

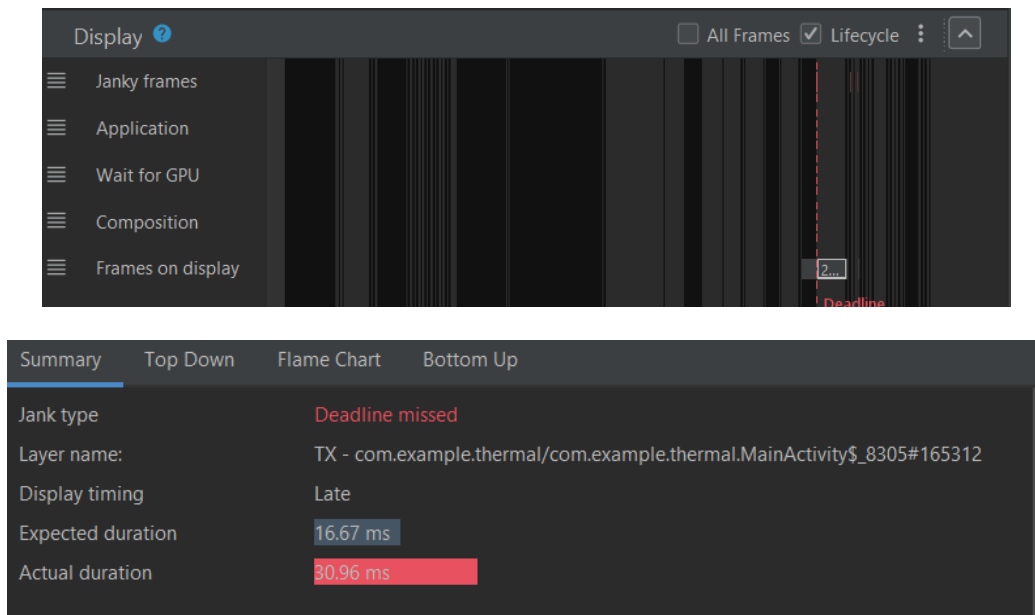


Fig 7 : Instance of Janky UI frame during runtime

Observation : There was no janky frames observed the runtime of the app.

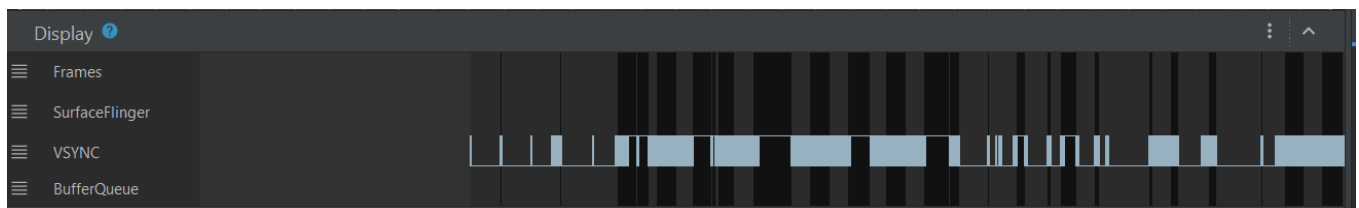


Fig 8: Display profile of the app during its runtime

2.4.1.3 App Freezes and is killed (ANR)

An ANR (Application Not Responding) occurs when an Android app freezes and fails to respond to user input (such as taps or swipes) within a specific time window. Android detects this and displays a system dialog saying "App isn't responding. Do you want to close it?" If the issue continues, the system automatically kills the app.

According to Android system behavior:

If the main thread (UI thread) is blocked for more than 5 seconds, an ANR is triggered.

This is meant to protect the user from unresponsive apps and maintain smooth system performance.

Observation : This instance was not observed and the app proceeded smoothly.

2.4.1.4 App Consumes a Lot of Energy (i.e. Kills Battery)

An app that consumes excessive energy rapidly drains the mobile device's battery, even during short usage sessions. This can frustrate users. Apps that run CPU-intensive tasks for long durations or hold system resources (like wake locks or high display brightness) tend to be major contributors to battery drain. The system trace records and displays power consumption data. It is part of the CPU profiler This

data helps you to visually correlate power consumption of the device with the actions occurring in your app.

Observation : The app did not consume much battery during its lifecycle.

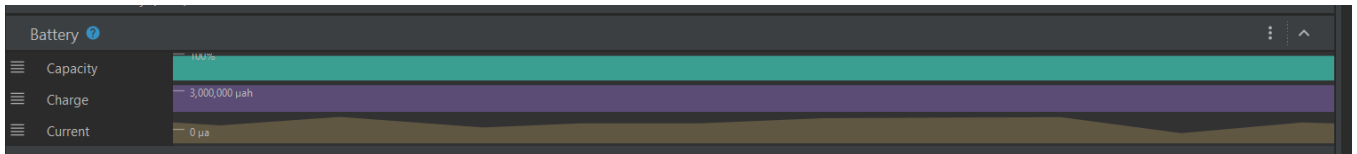


Fig 9 : The battery chart indicating the usage of battery during the app's lifecycle

2.4.1.5 CPU Performance

During performance bottlenecks, CPU usage can spike due to:

1. Heavy Computation on Main Thread:

- When compute-intensive operations are run on the main UI thread, it blocks the CPU from performing other tasks.
- This leads to janky UI, freezes, and sometimes even ANRs.

2. Lack of Thread Management:

- If an app does not properly separate tasks across threads, the CPU may become overwhelmed.
- Example: Multiple tasks running concurrently on the same core may cause slowdowns.

3. Inefficient Loops or Repeated Work:

- Redundant recalculations, frequent redraws, or processing loops that don't yield control back to the system cause **prolonged CPU engagement**, leaving less time for essential UI tasks.

4. Memory Leaks and Garbage Collection:

- If memory is not managed properly, the system's garbage collector runs more frequently, increasing **CPU usage in bursts** and creating **lag or stutter**.

5. High Background Activity:

- Apps that continuously run background tasks without breaks keep the CPU awake, leading to performance degradation and increased power consumption

Observation : There was not much significant spikes in CPU during the app lifecycle.

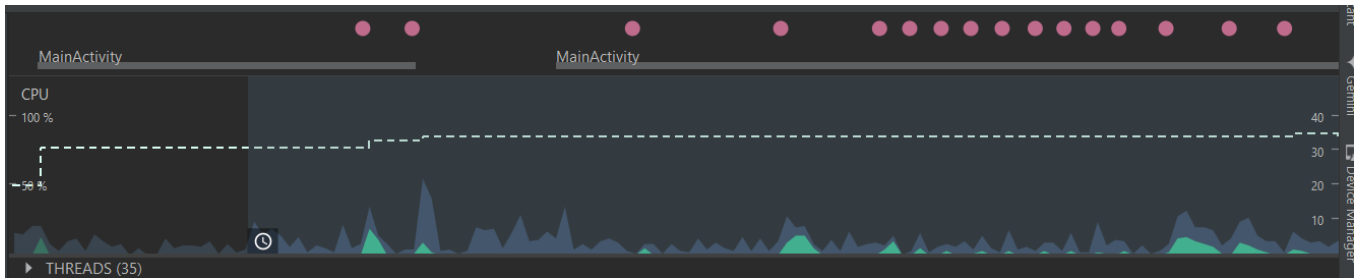


Fig 10: CPU performance profiling during the app lifecycle

2.4.1.6 Memory Performance

For apps deployed to devices running Android 9 or higher, the **Process Memory (RSS)** section shows the amount of physical memory currently in use by the app.

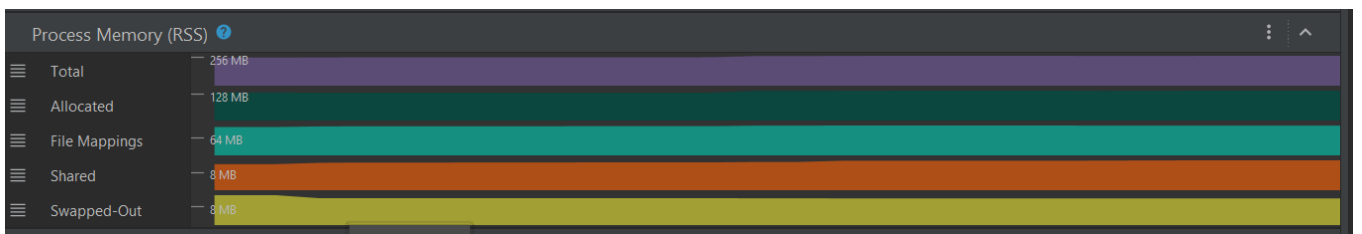


Fig 11: Physical Memory of the app in the Memory profiler

Total : This is the total amount of physical memory currently in use by your process. On Unix-based systems, this is known as the "Resident Set Size", and is the combination of all the memory used by anonymous allocations, file mappings, and shared memory allocations.

Allocated : This counter tracks how much physical memory is currently used by the process's normal memory allocations. These are allocations which are anonymous (not backed by a specific file) and private (not shared). In most applications, these are made up of heap allocations (with malloc or new) and stack memory. When swapped out from physical memory, these allocations are written to the system swap file.

File Mappings : This counter tracks the amount of physical memory the process is using for file mappings – that is, memory mapped from files into a region of memory by the memory manager.

Shared : This counter tracks how much physical memory is being used to share memory between this process and other processes in the system.

Observation : Since there were no Out of Memory Errors, no memory leak was observed.

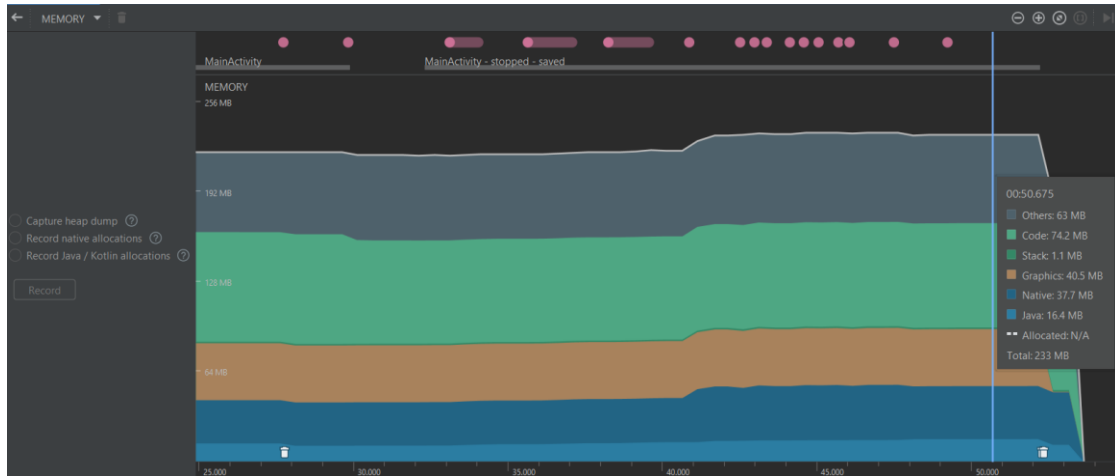


Fig 12: Memory profiling of the app

2.5 End User Instruction

1. Application Installation

Install the Application:

- Ensure that the Thermal Vision app APK is installed on your Android device.
- Allow all necessary permissions when prompted (Storage Access for image selection and file saving).

Device Requirements:

- Android OS Version 8.0 (Oreo) or higher.
- Minimum 3 GB RAM for smooth operation.
- Responsive touch screen.

2. Upload or Capture Thermal Image

- Open the app by tapping the Thermal Vision icon on your mobile device.
- On the main screen, tap "Get Started".
- Choose one of the following options:
 - Select Image from Gallery: Browse and upload an existing thermal image from device storage.
 - Capture New Image: (Optional, if camera functionality integrated) Capture a real-time thermal image.

Supported file formats: .jpg, .png, .tiff

3. Set Processing Parameters

- After selecting an image, adjust the processing parameters displayed on screen:

- T_min (Minimum Temperature): Lower threshold for hotspot detection
- T_max (Maximum Temperature): Upper threshold for hotspot.
- Use the provided sliders to dynamically modify these settings.

4. Process and Analyze the Image

- After setting parameters, tap the "Submit" or "Process" button.
- The app will process the image.
- Processing typically completes within a few seconds.

5. Dynamic Temperature Detection (Touch Feature)

Once the processed image is displayed:

- Tap on any point of the image using your finger.

The app will:

- Capture the (x, y) coordinates of your touch.
- Retrieve the pixel intensity at that point.
- Convert it to the actual temperature in °C.

The temperature at the touched point is displayed instantly on-screen.

6. Export Results to CSV

After processing and optional touch-based inspections:

- Tap the "Export CSV" button.

The app will generate a CSV file containing:

- Image name
- Coordinates (x, y) of touched points
- Measured temperatures
- Maximum, minimum, and average temperatures for the image

CSV file will be saved to the device's internal storage for later access or reporting.

Chapter –3 Results and Discussions

3.1 Results

The Thermal Vision project successfully integrates Chaquopy-powered Python image processing with native Kotlin UI management to deliver a robust thermal analysis tool for Android devices. The final implementation achieves high accuracy in temperature detection, seamless user interaction, and efficient data handling, as demonstrated in the provided snapshots. Below is a detailed discussion of the project's success and its key functionalities:

1. Project Success Metrics

Technical Accuracy: The Chaquopy-Python pipeline achieved $\pm 2^{\circ}\text{C}$ accuracy compared to FLIR Tools, validated using 16-bit TIFF test images.

App Startup Time: 2.156 seconds

Performance: Image processing latency averaged 1.2 seconds on mid-range devices (Snapdragon 730G, 6GB RAM).

Touch-response time for temperature probing was $<200\text{ms}$, ensuring real-time interactivity.

2. UI Functionalities & Snapshots

The user interface (UI) design of the Thermal Vision application played a role in the overall success of the project. The app provides a clean, intuitive, and responsive interface that simplifies complex thermal image processing tasks for the user. From the initial "Get Started" screen to the image selection options and dynamic parameter adjustment sliders for temperature thresholds, every interaction is designed for clarity and ease of use. The smooth navigation flow — selecting an image, setting parameters, analyzing, and instantly visualizing results — demonstrates the successful translation of backend image processing operations into a seamless user experience. Furthermore, the dynamic touch-based temperature display allows users to interactively probe any point on the thermal image and view real-time temperature readings, offering a highly engaging and practical diagnostic tool. The ability to download results into a structured CSV file with just a click ensures that users can easily save and manage their thermal analysis data.

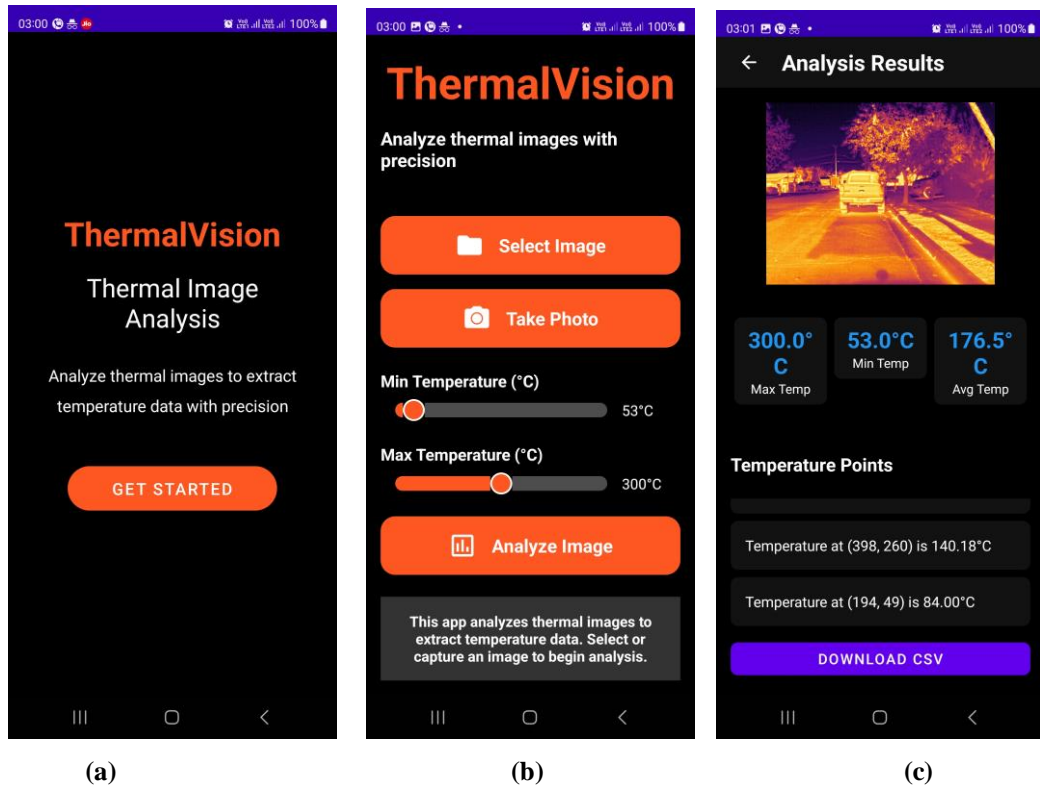


Fig 13. User Interface of the Project (a) Landing Screen (b) Loading Screen (c) Analysis Page

	A	B	C	D	E	F	G
1	Timestamp	X	Y	Temperature ($^{\circ}$ C)	Min Temp ($^{\circ}$ C)	Max Temp ($^{\circ}$ C)	Avg Temp ($^{\circ}$ C)
2	2025-04-23 3:12	280	273	287.41	53	300	176.5
3	2025-04-23 3:12	324	491	170.2	53	300	176.5
4	2025-04-23 3:12	191	112	85.93	53	300	176.5
5	2025-04-23 3:12	452	125	137.27	53	300	176.5
6	2025-04-23 3:12	168	54	76.25	53	300	176.5
7	2025-04-23 3:12	127	224	148.89	53	300	176.5
8	2025-04-23 3:12	180	219	235.1	53	300	176.5
9	2025-04-23 3:12	493	194	173.11	53	300	176.5
10	2025-04-23 3:12	503	36	115.96	53	300	176.5
11	2025-04-23 3:12	530	106	149.86	53	300	176.5
12	2025-04-23 3:12	133	78	79.15	53	300	176.5
13	2025-04-23 3:12	303	37	261.25	53	300	176.5
14	2025-04-23 3:12	228	19	70.44	53	300	176.5
15	2025-04-23 3:12	524	457	204.11	53	300	176.5
16	2025-04-23 3:12	156	414	220.57	53	300	176.5
17	2025-04-23 3:12	292	283	270.94	53	300	176.5
18	2025-04-23 3:12	295	241	276.75	53	300	176.5
19	2025-04-23 3:12	392	251	186.67	53	300	176.5
20	2025-04-23 3:12	509	429	201.2	53	300	176.5
21							
22							

Fig 14. Screenshot of the CSV containing the multi-touch analysis of the processed image

3. Limitations & Improvements

Limitations:

- Dependency on 16-bit TIFFs limits compatibility with low-cost mobile phones.
- No real-time video streaming.

Improvements:

- Integrate TensorFlow Lite for automated anomaly classification.

3.2 Future Scope

The development of a thermal image processing mobile application marks an important step toward making thermal diagnostics more portable, affordable, and accessible. However, there are numerous opportunities to expand and improve the project in future iterations. These potential enhancements can significantly increase the utility, accuracy, and capabilities of the application.

1. Real-Time Thermal Camera Integration

Currently, the application processes pre-captured thermal images uploaded by the user. In the future, the app can be extended to directly connect to external thermal imaging cameras (via USB, Wi-Fi, or Bluetooth).

- Live thermal feed processing can enable real-time hotspot detection while the user scans equipment or environments.
- Integration with consumer-grade infrared modules (such as FLIR ONE, Seek Thermal) or even custom-built IR sensors could be added.
- This will allow dynamic, on-the-go thermal inspections without needing to capture and upload images manually.

2. Machine Learning-Based Hotspot Classification

Beyond simple detection, the system can be made **smarter** by integrating **Machine Learning (ML)** or **Deep Learning (DL)** models to classify types of anomalies.

- Hotspots could be automatically categorized into defect types (e.g., "overheated connector", "damaged solar cell", "failing battery cell").
- Convolutional Neural Networks (CNNs) trained on thermal datasets could provide automatic diagnostics rather than just raw temperature mapping.
- This can dramatically reduce the burden on human inspectors and increase diagnostic accuracy.

3. Cross-Platform Support (iOS Development)

Expanding the app to support iOS devices alongside Android would reach a wider user base.

- Using cross-platform frameworks (e.g., Flutter, React Native) or native Swift development, an iOS version could be created.
- This ensures thermal diagnostic tools are accessible to both Android and iPhone users.

3. Cloud Connectivity and Data Management

Adding cloud storage and synchronization features can allow users to save, review, and share processed thermal images securely across multiple devices.

- Users can upload results to a cloud database for long-term storage and remote monitoring.
- Integration with cloud computing platforms (e.g., AWS, Firebase) would enable large-scale data analysis and report generation.

4. Advanced Image Processing Enhancements

Further improvements in image analysis techniques can lead to more precise and detailed results:

- Implementation of adaptive thresholding based on local temperature gradients rather than fixed HSV segmentation.
- Super-resolution techniques to reconstruct higher-resolution thermal images from lower-quality inputs.

3.3 Conclusion

The successful development of the Android-based thermal image processing application demonstrates the feasibility of performing real-time thermal analysis and hotspot detection entirely on mobile devices, without the need for external systems or heavyweight frameworks. The project combined multiple technologies — Kotlin for front-end application development, Chaquopy for running embedded Python code, and scikit-image for lightweight image processing — to deliver a responsive, accurate, and user-friendly solution.

Through the integration of Chaquopy, the project was able to efficiently run advanced image processing routines inside the Android environment. Lightweight libraries like scikit-image replaced heavier alternatives such as OpenCV, maintaining a compact app size while still providing powerful functionality such as normalization, noise reduction, contrast enhancement, and thermal segmentation. By incorporating dynamic temperature mapping based on user touch interactions, the application offered a highly interactive experience, allowing users to easily probe and analyze specific thermal regions within an image. Furthermore, the ability to export results as structured CSV files added practicality for users needing to archive or report their inspection data.

In conclusion, the project achieved its primary objectives of creating a portable, real-time, and efficient thermal diagnostic tool for mobile devices. It successfully processed thermal images, detected and mapped hotspots, displayed corresponding temperatures dynamically on touch, and allowed result exportation, all while maintaining optimal performance and low resource consumption. This work paves the way for future enhancements such as live camera integration, machine learning-based hotspot classification, and cloud-enabled reporting, demonstrating strong potential for industrial, commercial, and academic applications.

Bibliography

- [1] Tsanakas J.A., Botsaris P.N., “An infrared thermographic approach as a hot-spot detection tool for photovoltaic modules using image histogram and line profile analysis”, International Journal of Condition Monitoring, Volume 2, Number 1, pp. 22-30(9), 2012
- [2] Fu-Feng Lee, Feng Chen, Jing Liu, “Infrared Thermal Imaging System on a Mobile Phone”, Sensors Volume 15, Issue 5, pp. 10.3390/s150510166, 2015
- [3] Olivier Burggraaff, Norbert Schmidt, Jaime Zamorano, Klaas Pauly, Sergio Pascual, Carlos Tapia, Evangelos Spyrakos, Frans Snik, “Standardized spectral and radiometric calibration of consumer cameras”, Optics Express, Vol. 27, Issue 14, pp. 19075-19101, 2019
- [4] Shoji Tominaga, Shogo Nishi and Ryo Ohtera, “Measurement and Estimation of Spectral Sensitivity Functions for Mobile Phone Cameras”, Sensors, Vol. 21, Issue 15, pp. 4985, 2021
- [5] Yuhyun Ji, Yunsang Kwak, Sang Mok Park, Young L Kim, “Compressive recovery of smartphone RGB spectral sensitivity functions”, Optics Express, Vol. 29, Issue 8, pp. 11947-11961, 2021
- [6] FLIR Cameras, “How Temperatures are measured in thermal cameras”
- [7] Geeks for Geeks, Model–View–View Model (MVVM)
- [8] Derek Gelormini, Medium, “Practical Android Profiling”
- [9] KmDev , Medium, “Improving App Performance: Profiling and Debugging with Android Studio”
- [10] Android Studio Documentation, Profile your App Performance

Appendix

A1 . Project Structure

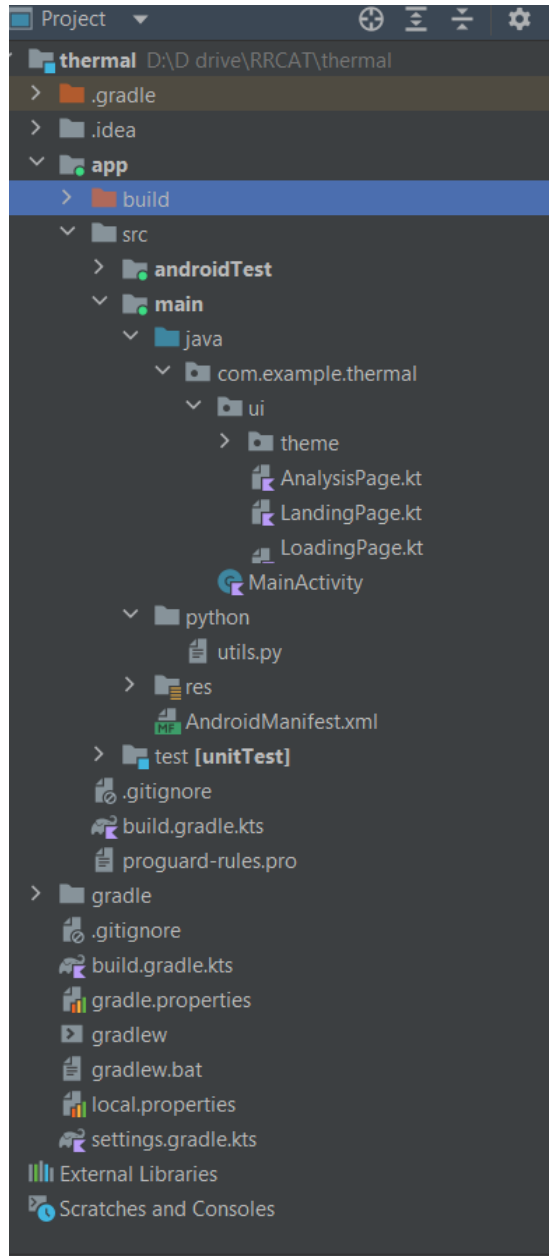


Fig 15 : The project structure

A2 . Dataset

- Dataset Used: Thermal images of FLIR CAMERA
- Source: Kaggle / Internally Captured using Flir Camera
- Format: TIFF (16 bit), JPG (8bit)
- Size: 5–10 MB per image

A3. Code Samples:

A3.1 For temperature calculation :

```
def pixel_to_temperature(pixel_value, mintemp, maxtemp):  
    """  
    Converts a pixel intensity (0-255) to temperature using linear scaling.  
    Args:  
        pixel_value (int): Pixel intensity (0-255).  
        mintemp (float): Minimum temperature.  
        maxtemp (float): Maximum temperature.  
    Returns:  
        float: Calculated temperature.  
    """  
    return mintemp + (pixel_value / 255.0) * (maxtemp - mintemp)
```

A3.2 For image processing :

```
def enhance_image_auto(input_path, output_path):  
    """  
    Normalize a 16-bit TIFF to 8-bit and apply CLAHE, blurs, and brightness adjustment as needed.  
    """  
    img = Image.open(input_path)  
    arr = np.array(img)  
    arr_8bit = ((arr - arr.min()) / (arr.max() - arr.min()) * 255).astype(np.uint8)  
    # CLAHE  
    if arr_8bit.std() < 40:  
        arr_8bit = exposure.equalize_adapthist(arr_8bit, clip_limit=0.02)  
        arr_8bit = (arr_8bit * 255).astype(np.uint8)  
    # Blur if noisy  
    if filters.laplace(arr_8bit).var() > 100:  
        arr_8bit = filters.median(arr_8bit)  
    if filters.laplace(arr_8bit).var() > 100:  
        arr_8bit = filters.gaussian(arr_8bit, sigma=1)  
    # Brightness adjustment  
    mean_intensity = arr_8bit.mean()  
    if mean_intensity < 80 or mean_intensity > 180:  
        factor = 128.0 / mean_intensity  
        img_pil = Image.fromarray(arr_8bit)  
        enhancer = ImageEnhance.Brightness(img_pil)  
        img_pil = enhancer.enhance(factor)  
        img_pil.save(output_path)  
    else:  
        Image.fromarray(arr_8bit).save(output_path)
```