

**STANDALONE PACKAGE DELIVERY VERIFICATION USING COMPUTER
VISION AND IMAGE METADATA EXTRACTION**

BY

CHOW YONG XIANG

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF COMPUTER SCIENCE (HONOURS)

Faculty of Information and Communication Technology

(Kampar Campus)

FEBRUARY 2026

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and appreciation to my supervisor, Ts. Dr. Tan Teik Boon, for providing me with the bright opportunity to engage in this software development project. It serves as a vital first step in establishing my career in software engineering. A million thanks to you.

I would also like to deeply thank Ts. Dr. Ooi Boon Yaik, also known as Dr. Alex, for exposing me to this opportunity and making the collaboration with a real-world industry partner, GDEX, possible. Thank you for your unwavering guidance, strategic foresight, and the direction you have provided in shaping my career path.

My profound gratitude extends to the management and operational teams at **GDEX**. A special mention goes to Ms. Anna for her exceptional administrative support, coordination, and trust in this project. Furthermore, I am incredibly thankful to the Kampar branch dispatch team and the driver who accommodated my live operational field testing. Your willingness to facilitate this industry-academic collaboration provided me with irreplaceable real-world data and logistical insights, bridging the gap between theoretical code and practical application.

To a very special person in my life, Dorothy Ling, for her patience, unconditional support, and love, and for standing by my side during hard times. Finally, I must say thanks to my parents and my family for their love, support, and continuous encouragement throughout the course.

COPYRIGHT STATEMENT

© 2026 Chow Yong Xiang. All rights reserved.

This Final Year Project report is submitted in partial fulfillment of the requirements for the degree of Bachelor of Computer Science (Honours) Information Systems Engineering at Universiti Tunku Abdul Rahman (UTAR). This Final Year Project report represents the work of the author, except where due acknowledgment has been made in the text. No part of this Final Year Project report may be reproduced, stored, or transmitted in any form or by any means, whether electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author or UTAR, in accordance with UTAR's Intellectual Property Policy.

ABSTRACT

Last-mile delivery is widely recognized as the most complex and expensive stage of the e-commerce supply chain, often characterized by high fragmentation and limited observability [1]. Challenges such as "porch piracy", the theft of unattended packages, and the lack of structured Proof-of-Delivery documentation lead to significant operational inefficiencies, including payment delays and unresolved customer disputes [2], [3]. This project addresses these vulnerabilities by introducing a **Zero-Trust Visual Verification Pipeline** delivered through a standalone **React Native** mobile application.

The system utilizes **Exchangeable Image File Format** metadata as a core novelty to enhance delivery verification. By extracting and validating high-precision geolocation and timestamp data at the point of capture, the application ensures that delivery evidence reaches a forensic-grade standard of precision [4]. This "Gated Capture" mechanism is further reinforced by a **Siamese Neural Network**, which provides temporal image validation by measuring the similarity between delivery photos to verify parcel identity across different stages [5], [6]. To maintain operational continuity in diverse environments, the architecture includes an adaptive accuracy algorithm for GPS-denied scenarios and a building detection model optimized for edge deployment.

Developed within the **Expo Application Services** ecosystem and supported by a **Supabase** cloud backend, this system provides a secure, tamper-resistant framework for logistics tracking. By binding parcel unique identifiers with verified environmental context, the proposed solution transforms delivery records into legally defensible assets, ultimately strengthening transparency and reducing fraudulent claims in last-mile logistics [7].

Area of Study: Computer Vision, EXIF Metadata

Keywords: Mobile Application, Proof of Delivery, EXIF Metadata, React Native, Machine Learning, Siamese Network, System Design

TABLE OF CONTENTS

STANDALONE PACKAGE DELIVERY VERIFICATION USING COMPUTER VISION AND IMAGE METADATA EXTRACTION.....	0
ACKNOWLEDGEMENTS.....	1
COPYRIGHT STATEMENT.....	2
ABSTRACT.....	3
TABLE OF CONTENTS.....	4
LIST OF FIGURES.....	7
LIST OF TABLES.....	10
LIST OF ABBREVIATIONS.....	11
CHAPTER 1 Introduction.....	13
1.1 Project Background Information:.....	13
1.1.1 Industry Example: GDEX.....	13
1.1.2 Challenges faced by Logistics and Courier Companies:.....	14
1.2 Problem Statement and Motivation.....	15
1.2.1 Problem Statement.....	15
1.2.2 Motivation.....	16
1.3 Research Objectives.....	17
1.4 Project Scope and Direction.....	18
1.4.1 System Boundaries and Core Functionality.....	18
1.5 Contributions.....	20
1.6 Report Organization.....	21
CHAPTER 2 Literature Reviews.....	23
2.1 Understanding Metadata: What is Metadata?.....	23
2.2 The Exchangeable eImage File Format - EXIF.....	24
2.3 Implementation of EXIF Metadata extraction.....	25
2.4 Compare and Contrast on Similar Applications.....	27
CHAPTER 3 System Methodology.....	32
3.1 General Development Methodologies.....	32
3.1.1 Phase 1: The Minimum Viable Product (MVP).....	33
3.1.2 Phase 2: Advanced Feature Development and Integration of Computer Vision Modules..	34
3.1.2 Phase 3: Operational Functions, Field Deployment, and Iterative Refinement.....	37
3.2 System Architecture Diagram.....	39
3.2.1 Layer 1: Edge Layer (Driver's Mobile Device).....	40
3.2.2 Layer 2: Cloud Layer (Supabase BaaS).....	41
3.2.3 Layer 3: Presentation Layer (Web).....	41
3.3 Use-Case Diagram and Use-Case Descriptions.....	42
3.3.1 Use-Case Descriptions.....	43
3.4 Activity Diagram:.....	52
CHAPTER 4 System Design.....	58
4.1 System Block Diagram.....	58

4.2 System Component Diagram.....	59
4.3 Data and Database Design.....	61
4.3.1 Data Dictionary.....	62
4.3.2 Entity Relationship Diagram.....	66
4.3.2 Related SQL Queries.....	67
4.4 Front-End Interaction & Capture Modules.....	69
4.4.1 Dual-Layer Overlay & Barcode OCR.....	69
4.4.2 POD Landing Page:.....	76
4.5 Verification & Metadata Logic.....	79
4.5.1 Geolocation Gating Algorithm.....	79
4.5.2 EXIF Metadata Injection Logic:.....	82
4.6 Machine Learning Model: Building Detection.....	86
4.6.1 Empirical Data Curation and Preprocessing.....	86
4.6.2 Transfer Learning via Neural Architecture Search (NAS).....	92
4.6.3 Edge Quantization for Mobile Development.....	95
4.6.3 Asynchronous Edge Integration (React Native).....	97
4.7 Deep Learning Neural Network: Siamese Network.....	100
4.7.1 Siamese Network Implementation.....	100
4.7.2 Pair-Based Dataset Design.....	101
4.7.2 Embedding Network Design.....	102
4.7.3 The Siamese Twin Architecture.....	104
4.7.4 Euclidean Distance Layer.....	104
4.7.5 Contrastive Loss.....	106
4.7.6 Training Configuration.....	107
4.7.7 Deployment Design.....	107
4.7.6 Web User Interface for Evaluation of Siamese Network.....	108
CHAPTER 5 System Implementation.....	110
5.1 Hardware Setup.....	110
5.2 Software Setup.....	111
5.2.1 Integrated Development Environment and Language.....	111
5.3 Development Environment Setup.....	112
5.3.1 Expo Application Service.....	112
5.3.2 Dependencies on Project Repository.....	114
5.4 System Operations.....	116
5.4.1 Dual-Layer Overlay and Barcode OCR for Structure Capture.....	116
5.4.2 Implementation of Geolocation Gating Algorithm.....	118
5.4.3 Real-Time ML Edge Building Detection.....	120
5.4.4 Injecting EXIF Metadata into the Captured Image.....	121
5.4.5 POD Landing Page.....	124
5.4.6 Validate Temporal Images with Siamese Network.....	126
5.5 Implementation Issues and Challenges.....	131
5.5.1 Build Environment Volatility and Configuration Management.....	131
5.5.2 Logistical Bottlenecks in Proprietary Dataset Curation.....	131
5.5.3 Domain Shift and the Semantic Gap in Operational Testing.....	132

5.6 Concluding Remarks.....	132
CHAPTER 6 System Evaluation, Field Testing, and Discussion.....	134
6.1 Verification of the EXIF Tags.....	134
6.2 System Assessment.....	135
6.3 Lab Evaluation of the Building Detection ML Model.....	135
6.3.1 Mathematical Evaluation Metrics.....	135
6.3.2 Performance Results and Confusion Matrix Analysis.....	136
6.3.3 Cloud Endpoint Deployment and Qualitative Edge-Case Verification.....	137
6.4 Evaluation and Testing of the Siamese Model.....	141
6.5 Live Operational Field Testing with GDEX.....	142
6.5.1 Objectives of Field Testing.....	142
6.5.2 Operational Scope & Logistics.....	142
6.5.3 Execution Methodology & Protocols.....	142
6.5.4 System Performance.....	144
6.5.5 Limitations and Future Improvements.....	148
CHAPTER 7 Conclusion and Recommendation.....	149
7.1 Conclusion.....	149
7.2 Research Limitations.....	150
7.2.1 Proof of Concept Status of the Siamese Network.....	150
7.2.2 Dependency on Historical Data and API Integration.....	150
7.2.3 Geographic and Environmental Constraints.....	151
7.2.4 Path to Full Autonomy.....	151
APPENDIX (TODO).....	152
Poster.....	152
Interfaces.....	153
/Utills.....	155
References.....	158

LIST OF FIGURES

Figure Number	Title	Page
Figure 1.1.1	Malaysia B2C E-commerce value projections	10
Figure 1.1.2	A Research Results on at what point consumers stop interacting with a brand they love	11
Figure 2.3.1	Python code written in Piexif library to extract the Metadata informations	22
Figure 2.3.2	Python code to implement coordinate conversion	22
Figure 2.4.2, 2.4.3	Geodetic Conversion of raw GPS data into Decimal Degrees	25
Figure 2.4.3	Latitude coordination precision by the actual cartographic scale purported	25
Figure 3.1.1	Iterative and Incremental Development Methodology Lifecycle in Software Systems	30
Figure 3.1.2	Terrace Housings Datasets collected for the Building Recognition Module	32
Figure 3.2.1	System Architecture Diagram	38
Figure 3.3.1	Use-Case Diagram depicting actors and interactions	41
Figure 3.4.1	Activity Diagram for Edge-Native Data Collection	51
Figure 3.4.1	Activity Diagram for The Decoupled Siamese Inference Pipeline	51
Figure 3.4.1	Activity Diagram for Liability Transfer & Client Retrieval	51
Figure 4.1.1	System Block Diagram	57
Figure 4.2.1	High Level Component Design Diagram	58
Figure 4.2.2	Camera Subsystem Component Design Diagram	58
Figure 4.2.2	Data Processing Component Design Diagram	58
Figure 4.2.4	Shared Types Component Design Diagram	58
Figure 4.3.1	A Screenshot of the Supabase Table Dashboard with records redacted	60
Figure 4.3.2	ERD Design of the Database	65
Figure 4.4.1	Example Consignment Note generated by GDEX	68
Figure 4.4.2	Flowchart of the Camera Capturing Procedure	68

Figure 4.4.3	Dashboard on Vercel for deploying the Landing Page	76
Figure 4.4.4	GitHub Repository of the Landing Page	76
Figure 4.6.1	Sample White and Brown parcel to be introduced in training image datasets	86
Figure 4.6.2	6 samples of the curated training datasets, with alternating flows between no parcels, brown parcels, and white parcels	86
Figure 4.6.3	Guidelines from GDEX on taking photo POD images	86
Figure 4.6.4	3 sample datasets collected for each categories	86
Figure 4.6.5	Example of unstandardized raw datasets uploaded to the training device	86
Figure 4.6.6	Different sub-folders of categorization of training datasets after pre-processing	86
Figure 4.6.2	Training dashboard from Vertex Google AI Platform	92
Figure 4.6.3	Training datasets quantity	92
Figure 4.6.4, 4.6.5	Results report and metrics of the trained model under NAS methodology	92
Figure 4.6.3	Model analysis on Neutron	95
Figure 4.6.4	Selected Conv2D node is with tensor uint8	95
Figure 4.7.1	Architecture pipeline of the implemented Siamese Network verification	100
Figure 4.7.2	One of the expanded dataset houses for training the Siamese Network	101
Figure 4.7.6	The User Interface design before CSS for the Siamese Network evaluations	108
Figure 5.3.1	Command Line interface that create initial Expo project repository	113
Figure 5.3.2	QR Code generated by EAS for direct testing on physical mobile device	113
Figure 5.3.3, 5.3.4	Dashboard UI of EAS	113
Figure 5.4.1	The in-app Screenshots of the Camera View under different backgrounds	116
Figure 5.4.2	Labels on-screen for GPS accuracy and Building Detection confidence score	116
Figure 5.4.3	The logical conditions for enabling capture mode	116
Figure 5.4.4	Visual representation of these states within the application's user interface	118

Figure 5.4.5	Error fallback the app is going to handle if GPS is not available	118
Figure 5.4.6	Building Recognition in Action	120
Figure 5.4.7	EXIF Metadata injection in Action	121
Figure 5.4.8	Share Sheet in Action, sending delivery notifications and POD Landing Page URL to recipients via WhatsApp	121
Figure 5.4.9	Local Storage of the POD image are intacted with EXIF Metadata	121
Figure 5.4.10	Screenshot of the POD Landing Page	124
Figure 5.4.11	Hasher Checking of the SHA-256 done on image EXIF	124
Figure 5.4.12	Clicking on the coordinates directs users to Google Maps	124
Figure 5.4.13, 5.4.14	In-app screenshot of the Single Comparison feature	126
Figure 5.4.15	In-app screenshot of the Batch Testing feature	126
Figure 5.4.16, 5.4.17, 5.4.18, 5.4.19	In-app screenshot of the Auto-Pairing feature	126
Figure 5.4.20	In-app screenshot of images uploading to Model Validation tab	126
Figure 5.4.21, 5.4.22, 5.4.23	Reporting and analysis done in Model Validation tab	126
Figure 6.1.1	Interface of the external website that read image EXIF data	135
Figure 6.3.1	Evaluation details report shown in Google Vertex AI Platform	137
Figure 6.3.2	Confusion matrix of the Building Detection model	137
Figure 6.3.3	Front facing housing and performance of the model	138
Figure 6.3.4	Performance of the model on half housing and half foyer	138
Figure 6.3.4	Performance of the model on building foyer	138
Figure 6.3.5	Performance of the model on a clear sky, road view Image	138
Figure 6.3.6	Performance of the model on non-building architecture	138
Figure 6.5.1	Ground truth options after capturing in Field Test mode	143
Figure 6.5.2, 6.5.3	Report data captured on Field Test	145
Figure 6.5.4, 6.5.5	Confusion Matrix and KPI table of the data captured in Field Test	145
Figure 6.5.6	Interactive map tailored for analysis of the Field Test route	145

LIST OF TABLES

Table Number	Title	Page
Table 2.1	Different Types of Metadata, Example Properties, and Primary Usage	20
Table 2.4.1	Comparison of the most well-known POD system in the current market	25
Table 3.3.1	The overall view of Use Case Descriptions listed out in this section, including their ID, Name, Primary Actor, and Purpose	42
Table 3.3.2	Use-case description for UC01 Capture POD Image	42
Table 3.3.3	Use-case description for UC02 Execute Siamese Inference Service	42
Table 3.3.4	Use-case description for UC03 Access POD Portal Landing Page	42
Table 3.3.5	Use-case description for UC04 Dispatch delivery notifications	42
Table 4.3.1	EXIFMetadata injected in each POD image	61
Table 4.3.2	Table of ‘delivery_records’ relation stored in Supabase Backend	61
Table 4.3.3	Table of ‘field_test_records’ relation stored in Supabase Backend	61
Table 4.3.4	Table of Supabase Storage Buckets data design	61
Table 4.4.1	Data Integration design of the Camera View	68
Table 4.4.2	Visual Design of the POD Landing Page	76
Table 4.5.1	Accuracy Threshold Decision Table	79
Table 4.5.2	Data Integration design of the geolocation algorithm	79
Table 4.5.3	Table show field mapping of the value sources to EXIF tags	82
Table 4.5.4	Data Interaction design of the EXIF injection algorithm flow	82
Table 5.1	Specifications of the Laptop as development workstation	110
Table 5.2	Specifications of the mobile device for development and debugging	110
Table 5.2.1	Specifications of the Software and Development Tools	111
Table 5.3.2	Core Dependency Table	114

LIST OF ABBREVIATIONS

Short Form	Long Form
API	Application Programming Interface
B2C	Business-to-Consumer
BaaS	Backend As A Service / Backend as a Service
CAGR	Compounded Annual Growth Rate
CI/CD	Continuous Integration/Continuous Deployment
CNN	Convolutional Neural Network
CSS	Cascading Style Sheets
CV	Computer Vision
DD	Decimal Degrees
DMS	Degree, Minutes, Seconds
EAS	Expo Application Services
EER	Equal Error Rate
EXIF	Exchangeable Image File Format
FAR	False Acceptance Rate
FN	False Negatives
FP	False Positives
FRR	False Rejection Rate
GNSS	Global Navigation Satellite System
GPS	Global Positioning System
HUD	Head-Up Display
IDE	Integrated Development Environment
IFD	Image File Directory
JSI	JavaScript Interface
ML	Machine Learning
MVP	Minimum Viable Product
NAS	Neural Architecture Search
NPU	Neural Processing Unit

OCR	Optical Character Recognition
OS	Operating System
POD / POD	Proof of Delivery / Proof-of-Delivery
PR AUC	Average Precision / Area Under Curve
QR	Quick Response
SDK	Software Development Kit
SOP	Standard Operating Procedure
TN	True Negatives
TP	True Positives
UID	Unique Identifier
URL	Universal Resource Locator

CHAPTER 1 Introduction

1.1 Project Background Information:

E-commerce, refers to the online buying and selling of goods, service, and data over the Internet and other electronic devices[8]. Especially after COVID-19, the shift from physical purchase to online purchasing of goods has been significant. Malaysia is ranked as one of Southeast Asia's top e-commerce markets, and is projected to rise steadily from 2024 to 2029 [1]. Especially on the Business-to-Consumer (B2C) segment market in Malaysia will grow a Compounded Annual Growth Rate (CAGR) of 17% until the year 2023, stated by [2]

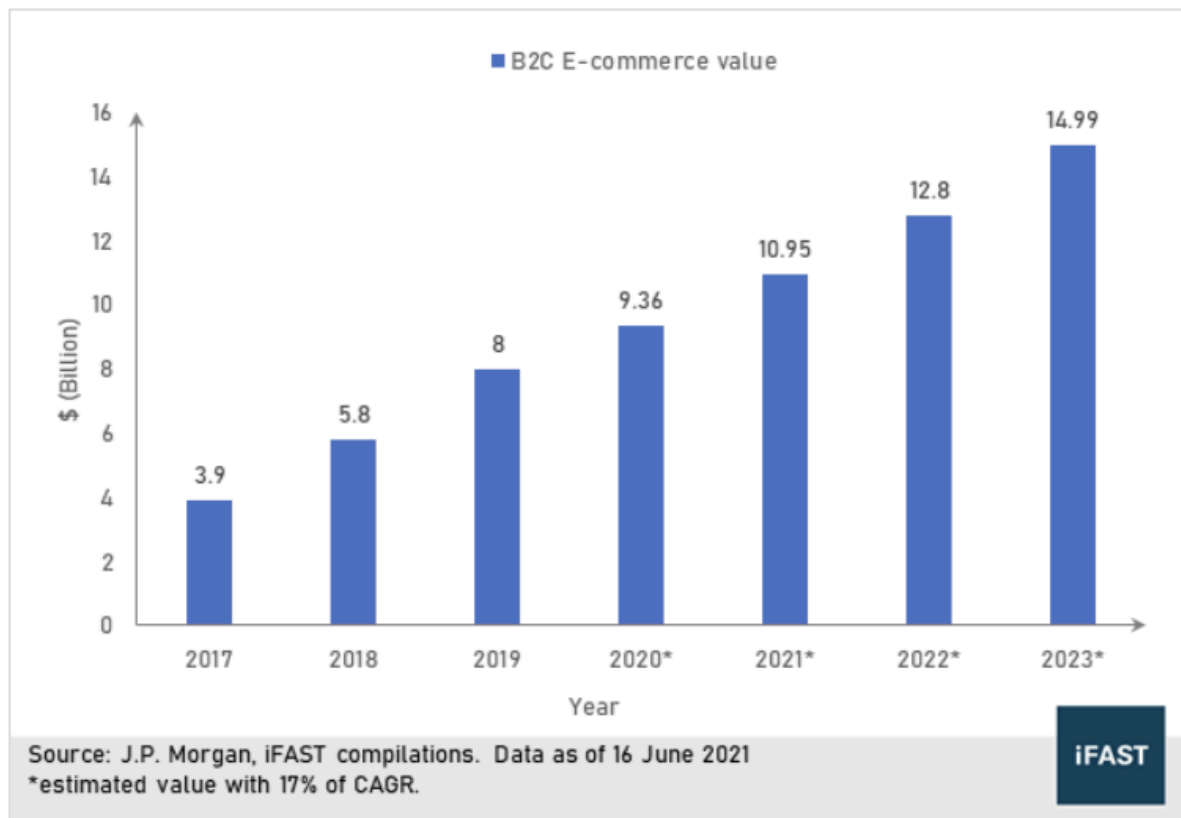


Figure 1.1.1 Malaysia B2C E-commerce value projections

1.1.1 Industry Example: GDEX

GDEX is one of the Malaysian leading logistics and courier companies that are in prime position to continue capturing the future growth in this industry. During the COVID-19

pandemic, it has done a 14.5% growth in express delivery revenue (2019–2020), and 40.7% growth in the logistics segment (2020). It is obvious to see that companies like GDEX highlight how e-commerce growth directly increases the burden on courier systems. The growth proves that the expectations of demands increased within the express and logistics service from these online business activities. [2]

1.1.2 Challenges faced by Logistics and Courier Companies:

According to Statista [3], more than 75% of consumers prioritize secure delivery and tracking as the most important element in the e-commerce delivering process. Therefore, logistics companies like GDEX should decide the systematic and structured approach in optimizing their delivery process.

According to a survey done by PwC, one out of every three customers [4] will stop shopping at their favourite store after just one bad experience. It is important for businesses to make sure their delivery system is easy, effective, and enjoyable for customers.

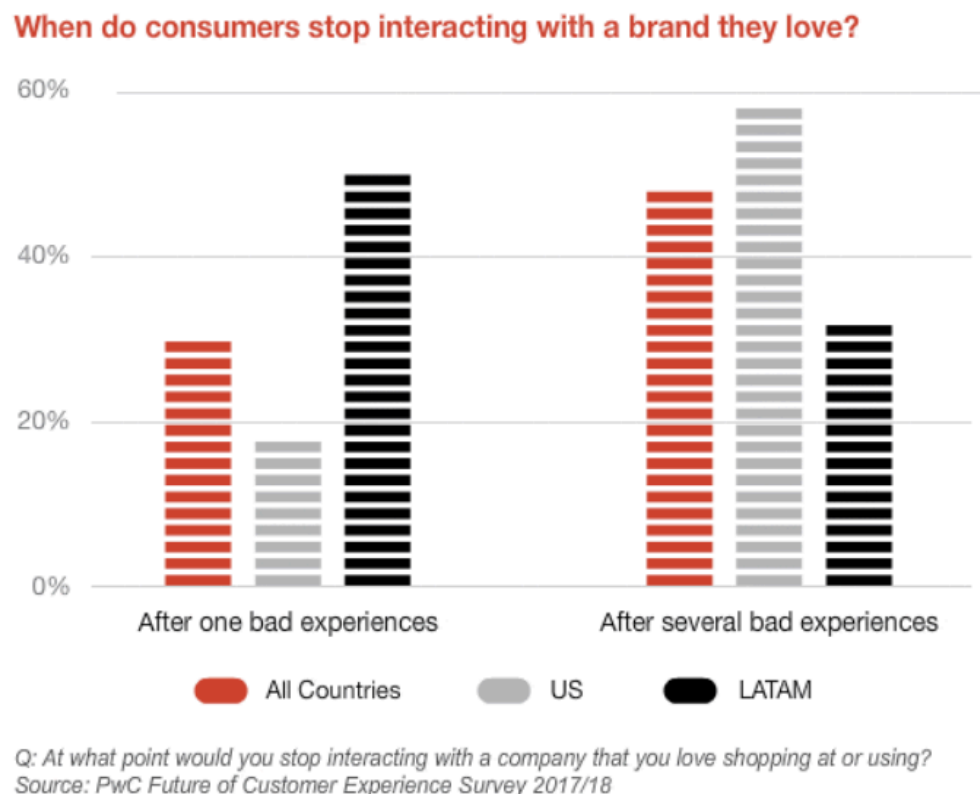


Figure 1.1.2 A Research Results on at what point consumers stop interacting with a brand they love

POD systems refers to the documentation that validates the customers have received their goods. Many organisations nowadays have moved to electronic POD (ePOD) to help automate and increase efficiency of the process [5].

In modern digitalized POD systems, it enables logistics teams such as drivers, technicians, and other field personnel [6] to quickly capture and retrieve order records with associated proof such as Photos, Digital Signatures, Timestamps, Geolocation data and more.

The use of POD systems increases accuracy by eliminating disputes via captured time and geolocation data. It increases transparency by having a centralized system for internal tracking. And also reduce paperworks, speeds up confirmation and record-keeping processes, thus enhancing operations efficiency. Parcel delivery companies, such as GDEX, need to integrate such systems to prove that a package was delivered at the correct place and time.

1.2 Problem Statement and Motivation

1.2.1 Problem Statement

Unstructured and Fragmented Proof Of Delivery Image Acquisition

The existing legacy process requires couriers to capture multiple separate photographs, one for the parcel and another for the delivery environment. However, POD imagery is highly dependent on human intervention, frequently resulting in unstructured outputs such as blurry, low-quality captures that lack environmental context. Without a unified capture interface, not only this fragmented workflow increases time spent at each delivery point, it also introduces inconsistencies in how visual evidence is recorded across the logistics network, such as the spatial relationship between the parcel and the building.

Weak Forensic Traceability and Absence of Mathematical Certainty

Standard POD images are often stored as raw files without intrinsic, verifiable metadata. Without embedded digital fingerprints such as geolocation coordinates and synchronized timestamps, these images lacks the forensic weight and mathematical certainty required to verify delivery authenticity with absolute precision. The weak forensic traceability complicates auditing processes, particularly resolving customer disputes. Moreover, the

absence of structured metadata diminishes the image's overall value as a verifiable digital evidence, reducing the credibility of proofs submitted to the final customer.

Lack of Cross-Temporal Location Authenticity Verification

A critical gap exists in the industry's inability to perform cross-temporal authentication, where a current delivery capture is verified against historical data of the same location. This creates an exploitable vulnerability where mistaken deliveries at visually similar but incorrect locations goes undetected. Currently, the only mechanism to mitigate this issue is through manual image auditing, where officers must reconcile thousands of daily PODs to ensure completeness and correctness [9]. This manual pipeline is resource-intensive, prone to human error, and inherently non-scalable, resulting in operational bottlenecks and frequent "second delivery attempts" when real-time location verification fails.

1.2.2 Motivation

Logistics companies typically are linked to a process called Last-mile delivery. It is the final stage where parcels are moved from the Logistic Hub to the Customer, and is widely regarded as the most complex, costly, challenged stage of logistics [7]. The increasing demand of Last-Mile Delivery has put significant challenges on last-miles companies.

The Last Mile Delivery has the highest chance of mistakes such as wrong address or misplaced parcels. Customers tend to face the problem of stating "I did not get my parcel", creating tension with the company. Since Last Mile Delivery is the only stage where customers directly see, it serves as a unique and significant stage where trust is built between customers and the company. However, blurry photos, vague GPS tags, and unstructured proofs of delivery reduce customer confidence, even if the parcel was delivered to the correct location [7]. Therefore, there is a need for trust and transparency.

As customers' expectations are rising, the standards are set by Logistic and Courier giants like Amazon Prime and Walmart. By using effective POD systems, it gives customers a peace of mind and gives the company a competitive advantage, since a structured approach serves the purpose of reassuring customers. In Malaysia's growing e-commerce market, companies that could achieve the best clarity, proof, and trust in Last Mile Delivery could stand out significantly against larger competitors.

Moreover, there have been cases where customers make false claims that they have not received the package [8][9]. In the case of business disputes, having a weak POD system positions the business to face legal issues, additional financial disputes, and reputation at risk [10].. By failing to offer strong legal protections, it leaves business vulnerable to fraudulent claims and lawsuits. Alternatively, if a customer has truly not received their package [11] an unreliable POD system makes it hard to find cause of such discrepancy, especially if it is delivered to the wrong address. The process of reverse engineering, and reverse tracking tend to be complicated, and require a lot of time and professional efforts.

With the issues of Last Mile Delivery, this paper proposes a method for modern POD systems to leverage forensic-style analysis, specifically EXIF (Exchangeable Image Files) Metadata that are stored within image files to strengthen credibility, customer trust, competitive advantage, and serve as a more robust legal proof. Captured in a structured manner, when extracted and verified, EXIF acts like a digital fingerprint, adding an extra layer of trustworthiness to the POD evidence.

1.3 Research Objectives

The primary goal of this research is to develop an intelligent, standalone package delivery verification system that transitions the proof-of-delivery process from a manual, error-prone workflow to an automated, forensically verifiable pipeline, especially in the Last Mile Delivery Phase. The specific objectives are as follows:

1. **To design and develop a structured Dual-Layer Head-Up Display (HUD) capture interface** that utilized computer vision (CV) to simultaneously recognize parcel barcodes and building structures. The computer vision functions are achieved via a fine-tuned Machine Learning (ML) model. This objective aims to enforce a standardized capture protocol that simultaneously consolidates and records both the delivery environment (building context) and the specific parcel identifier in a unified visual frame. The approach directly addresses the current limitations in forensic traceability by embedding crucial contextual information directly within the visual evidence [\[10\]](#).
2. **To implement an Exchangeable Image File Format (EXIF) based Metadata Injection Pipeline** capable of embedding high-precision geolocation and temporal

data directly into the captured delivery imagery header. This objective focuses on establishing mathematical certainty and forensic traceability, transforming raw image files into verifiable digital evidence that resists the ambiguities typically found in legacy POD systems [9].

3. **To develop a Cross-Temporal location authenticity verification module** utilizing a Siamese Network deep learning architecture. Whenever a new POD image is uploaded, the system should retrieve historical images based on the geolocation extracted from the current image to verify that they depict the same building. This objective seeks to mitigate mistaken deliveries by automatically comparing real-time delivery captures against historical baseline images of the destination, thereby providing an intelligent layer of verification that does not rely on manual auditing or resource-intensive human reconciliation [9].

1.4 Project Scope and Direction

The scope of this project encompasses the development of a comprehensive Proof of Delivery mobile application designed for logistics operations, specifically in partnership with GDEX. The system integrates EXIF metadata injection, geolocation gating, computer vision-based barcode recognition, and deep learning-powered location verification (in both Building Recognition and Siamese Network Validation) to transform traditional manual delivery workflows into an automated, forensically verifiable pipeline.

1.4.1 System Boundaries and Core Functionality

The developed system operates within the following boundaries:

In-Scope Features:

- **Structured Image Capture:** Dual-Layer HUD interface that enforces standardized photo composition via real-time overlay guides, simultaneously capturing parcel barcodes and building structures in a unified frame. The upper layer provides framing guides for the background building, while the lower layer guides capture of the physical parcel with its visible consignment note.
- **Geolocation Gating Algorithm:** An adaptive GPS accuracy verification mechanism that enforces a high-precision geolocation capture (configurable threshold of <10m)

with a time-bounded (e.g., 5s timeout) fallback for GPS-denied environments using network-based or last-known positioning.

- **EXIF Metadata Injection Pipeline:** Client-side embedding of geolocation coordinates (latitude/longitude), timestamps, and device metadata (e.g., Make, Model, Orientation) directly into JPEG image EXIF headers, creating tamper-evident digital evidence, and plays the role of a Future Reference Point for the Siamese Network Verification.
- **Computer Vision Parcel Identification:** Computer vision based real-time barcode localization, integrated with Optical Character Recognition (OCR) for automated parcel Unique Identifier extraction.
- **Edge-Native Semantic for Building Recognition Validation:** The component encompasses a lightweight, edge-native building recognition module designed to filter visual data prior to database submission. Utilizing transfer learning, this module is fine-tuned on the **MobileNetV3** foundational architecture. The model was trained using a specifically curated dataset of 578 terrace housing building images in a typical Malaysia residential area. The model functions strictly as a real-time semantic gatekeeper running directly on the mobile device's CPU.
- **Deep Learning Cross-Temporal Location Verification:** The development and implementation of Siamese Deep Neural Network trained on close to 100 distinct terrace housing building in a Malaysia residential area, the pairing generated more than 1000 datasets to be used as training sets. The Deep Neural Net computes high-dimensional feature distances between paired visual inputs. The scope of this module is to compare newly captured POD images against a database of historical, geolocated baseline building images to confirm delivery location authenticity and prevent misdeliveries [\[11\]](#). This module is implemented as a **functional independent Web UI**, demonstrating the system's ability to verify current delivery images against a historical baseline.
- **Foundational Infrastructure:** Following the Siamese Network Feature above, this features describe the development of a "Grounding Infrastructure" that links EXIF metadata as a "Foreign Key" to query historical data—a necessary prerequisite for autonomous logistics.
- **Asynchronous Client Notification & Verification Portal:** Leverages native Operating System (OS) share sheet protocols to interface directly with third-party messaging platforms (primarily WhatsApp). The system dynamically generates and

transmits a secure landing page URL, granting the recipient immediate access to view the verified delivery image and its associated spatial metadata.

Out-of-Scope / Limitations:

- **Real-Time Live Database Integration:** Automated real-time retrieval of historical POD data during active delivery operations is currently out of scope. This functionality requires a full-scale API integration with GDEX's proprietary historical database, which contains millions of records beyond the scope of this project.
- **Live Operational Autonomy:** While the Siamese Network is functional, the "Fully Autonomous" trigger, where the AI makes a real-time decision to authorize a drop without any human oversight is treated as a **Proof of Concept**.
- **Nationwide Deployment:** The system is optimized for residential environments and was tested using a curated dataset of approximately 1,000 housing structures rather than a nationwide industrial rollout.

1.5 Contributions

This project introduces a paradigm shift in logistics Proof of Delivery (POD) by moving away from manual human auditing with Artificial Intelligence driven verification. This comprehensive system significantly enhances the accuracy, efficiency, and trustworthiness of last-mile delivery operations, thereby contributing to the broader goal of sustainable urban logistics [\[12\]](#). The novelty of the system lies in the fusion of three distinct technologies into a unified **Zero-Trust Visual Verification Pipeline**:

1. Edge-Native Semantic Validation (Building Recognition)

- The system introduces a proactive, edge-native gatekeeper via Deep Learning model (MobileNetV3) directly onto the mobile CPU. Which is a huge contribution compared with current industry standards that accept any raw visual inputs and rely on manual back-office review for dispute resolution. By forcing the system to semantically validate that the camera frame contains a building *before* capture, it completely prevents fraudulent or erroneous data from ever entering the database.

2. EXIF Metadata Injection into POD Images

- **The Mechanism:** The application intercepts the verified image and permanently binds hardware-level EXIF metadata (GPS coordinates, altitude, heading, and milliseconds) directly into the file. It elevates a standard POD image from a subjective photograph into an **objective, mathematically verifiable data node**. It creates an immutable digital anchor that ties a mathematical link between a physical location and a digital image. This ensures that every POD is not just a photograph, but a forensically traceable digital asset.
- **3. Cross-Temporal Visual Geofencing (The Siamese Network)**
 - **The Mechanism:** The architecture leverages the EXIF metadata to query a historical database, fetching a baseline image of that specific coordinate. A Siamese Neural Network then calculates the high-dimensional feature distance between the new POD and the historical baseline. This creates an **infinitely scalable, self-improving verification loop**. Overtime, the system is able to build and construct a cluster of visual maps that accurately represent diverse delivery locations node.

While EXIF, Convolutional Neural Networks (CNN) for video frames processing, and Siamese Networks exist independently, this research is novel in its **Architectural Synthesis**. By linking these three distinct layers, the integrated approach significantly mitigates the risk of tampering and fraudulent claims in last-mile delivery logistics [\[13\]](#), [\[14\]](#), [\[15\]](#) , while also providing a grounding infrastructure needed for achieving fully autonomous logistic delivery in the future.

1.6 Report Organization

This research report is structured into seven chapters, guiding the reader through the conceptualization, design, implementation, and empirical evaluation of the standalone package delivery verification system. The chapters are organized as follows:

- **Chapter 1: Introduction** – This chapter establishes the project background, using GDEX as a primary industry example to highlight current challenges in logistics. It defines the problem statement, research objectives, and the project's scope, providing

a foundational understanding of the system's intended contribution to the courier industry.

- **Chapter 2: Literature Review** – This chapter explores the technical foundations of the research. It focuses on the nature of metadata, specifically the Exchangeable Image File Format, and examines existing methodologies for metadata extraction. It also provides a comparative analysis of similar applications to identify current technological gaps.
- **Chapter 3: System Methodology/Approach** – This chapter details the phased development lifecycle of the project, from the Minimum Viable Product to advanced computer vision integration. It presents the three-tier system architecture and utilizes use-case and activity diagrams to map out system processes.
- **Chapter 4: System Design** – Chapter 4 serves as the technical blueprint of the system. It covers database design, front-end interaction modules such as Barcode OCR, and the logic behind geolocation gating. Significant focus is placed on the design of the machine learning components, including the Building Detection model and the Siamese Network architecture for image verification.
- **Chapter 5: System Implementation** – This chapter documents the physical and digital realization of the system. It details the hardware and software environment and describes the operational implementation of core features. It also discusses the practical challenges encountered during development, such as dataset curation and environment volatility.
- **Chapter 6: System Evaluation, Field Testing, and Discussion** – This chapter provides a rigorous assessment of the system's performance. It includes mathematical evaluation metrics for the machine learning models, confusion matrix analysis, and qualitative results from live operational field testing conducted with GDEX. The chapter concludes with a discussion on the system's operational limitations.
- **Chapter 7: Conclusion and Recommendation** – The final chapter summarizes the research findings and reflects on the project's objectives. It acknowledges research limitations regarding the Siamese network and geographic constraints while proposing a strategic path toward full autonomy and future system improvements.

CHAPTER 2 Literature Reviews

2.1 Understanding Metadata: What is Metadata?

Metadata is pervasive in our modern informatic systems. By definition, it encapsulates the information about the item's creation, name, topic, features, hence sometimes called the statement "Data of Data(s)". According to Jenn Riley[\[16\]](#), the core features of most software packages we use everyday are metadata-driven. It is the key to the functionality of the systems, enabling the person of interest to find essential information about this data once extracted. [\[17\]](#)

Metadata Type	Example Properties	Primary Uses
Descriptive metadata	Title Author Subject Genre Publication date	Discovery Display Interoperability
Technical metadata	File type File size Creation date/time Compression scheme	Interoperability Digital object management Preservation
Preservation metadata	Checksum Preservation event	Interoperability Digital object management Preservation
Rights metadata	Copyright status License terms Rights holder	Interoperability Digital object management
Structural metadata	Sequence Place in hierarchy	Navigation
Markup languages	Paragraph Heading List Name Date	Navigation Interoperability

Table 2.1 Different Types of Metadata, Example Properties, and Primary Usage

Firstly, Descriptive Metadata stores the information for finding or understanding a resource. For example the title, author, subject and keywords of a book.

Next, we have Administrative Metadata, that serves the purpose of managing resources. Within this umbrella, it can be further divided into 3 sub-categories, namely:

- **Technical Metadata:** that contains file types, formats, and info to decode or render files
- **Preservation Metadata:** for long-term management, such as checksums, migration or emulation
- **Rights Metadata:** can be used for the protection of intellectual property, and licenses (such as the Creative Commons License)

Thirdly, is Structural Metadata that describes relationships of parts of a resource. For example, the sequence of pages, table of contents and more.

Lastly, is Markup Languages, which mixes metadata and content with flags or tags. It marks structural and semantic features such as paragraphs, formatting, or semantic info. [\[18\]](#), [\[19\]](#)

2.2 The Exchangeable eImage File Format - EXIF

According to the paper, the Exchangeable Image File Format (EXIF) is a tag structured for embedding metadata into image files. It originated from the Japanese digital camera industry, and is now supported by nearly all digital cameras and mobile smart phones. [\[20\]](#)

On a technical side, the file formats that EXIF data are compatible with are TIFF and JPEG. It is notable that EXIF does not support JPEG2000, PNG, or GIF files. Social media platforms such as Instagram or Facebook often stripped off EXIF data of a post, but it is still possible that the Image embedded geolocation has been read and stored before removed.

EXIF stores multiple types of metadata that listed below:

- Technical metadata such as pixel dimensions, ISO settings, aperture, lens info and more
- Administrative Metadata such as data and time taken

- Geospatial Metadata: which are the Global Positioning System (GPS) coordinates of where the Image was taken

In the context of POD systems, EXIF data acts as the technical metadata that validates When, Where, and with What device a delivery photo was taken, hence serving as a digital forensic evidence.

2.3 Implementation of EXIF Metadata extraction

Implementation of Multi-Format EXIF Metadata Extraction from Images is a research paper where the authors use Python libraries, specifically Piexif and Pyheic to extract metadata from images. The paper also states geolocation format conversion, from raw low-level data and converting into readable format by humans, and integrating the GPS coordinates with Google Maps URL.

- How to use the Python library: Piexif and Pyheic to get metadata from image
- Geolocation Format Conversion → Google Maps integration

There are five main image file directories (IFDs) into which the metadata is organized: Primary, EXIF, Interoperability, Thumbnail, and GPS. In social media, EXIF metadata is often used to automatically sort or tag images by location, date, or device. The EXIF metadata markers are classified into several distinct categories or folders known as IFDs (Image File Directories).

According to the author, common applications of EXIF data include storing and retrieving the information of geolocation, enhancing copyright, processing image enhancement, conducting forensic analysis, and classification.

However, image files are typically stored in multi-format format such as JPEG, HEIC, HEIF and others. Apart from scalability and performance issues with diversifying large datasets, the handling of unsupported, missing, or corrupted metadata is a serious problem to be tackled in order to process efficiently and resource effectively. There are limitations with our existing tools due to often being format specific. Therefore, the paper suggests a more flexible, multi-format approach to metadata extraction to support a wider range of applications.

Traditional methods use tools like ExifTool, and Pillow. However, the author suggests the use of modern libraries like Piexif for interpreting EXIF data from JPEG file formats, and Pyheif to read from HEIC/HEIF format image files then process it with Piexif.

```
get_exif_info_from_jpeg(image_path):  
  
    try:  
        exif_dict = piexif.load(image_path)  
        gps_info = exif_dict.get("GPS", {})  
        date_time_info = exif_dict.get("Exif", {}).get(piexif.ExifIFD.DateTimeOriginal, b").decode('utf-8')  
  
def get_exif_info_from_heic(image_path):  
  
    try:  
        heif_file = pyheif.read(image_path)  
        metadata = heif_file.metadata or []
```

Figure 2.3.1 shows the Python code written in Piexif library to extract the Metadata informations

In this paper, the author states that the extracted geolocation information from the EXIF data can be combined with file annotations. Coordinate Conversion or Geodetic Conversion, is where this GPS metadata is further converted into human readable decimal format for generating a Maps Link.

```
lat_ref=gps_info[piexif.GPSIFD.GPSLatitudeRef].decode("utf-8")  
lon_ref=gps_info[piexif.GPSIFD.GPSLongitudeRef].decode("utf-8")  
lat_values = gps_info[piexif.GPSIFD.GPSLatitude]  
lon_values = gps_info[piexif.GPSIFD.GPSLongitude]
```

Figure 2.3.2 shows the Python code to implement coordinate conversion

Figure above shows that the latitude and longitude that describes a location on Earth's surface are stored as arrays of rational numbers in the order of degrees, minutes, and seconds.

By having the coordinates in decimal format, it is then plotted on a map and generates a Google Maps URL. This conversion step is crucial in enabling the retrieved data to be accurate enough and usable for further analysis or mapping, such as forensic investigations, or used for our Proof of Delivery system.

2.4 Compare and Contrast on Similar Applications

System	Features	Strengths	Weaknesses
Routific	Route optimization, delivery notifications, POD with photos/signatures	Improves efficiency, transparency	No structured photo rules, GPS verification limited
Track-POD	Real-time performance tracking, signature/photo/ barcode capture	Cloud storage, full POD record	Dependent on internet/cloud, weak parcel-to-location verification
Onfleet	Driver tracking, auto-dispatch, customer notes	Strong logistics optimization, real-time visibility	POD limited to notes/photos, not robust against disputes
OptimoRoute	Route planning with time windows, vehicle capacity	Efficient scheduling, accurate ETAs	Not designed for POD validation, only logistics
Route4Me	Real-time tracking, multi-stop optimization	Reduces fuel/time costs	Focus on routing, not POD validation
OnTime 360	Dispatching, GPS tracking, notifications, signature capture	Complete delivery management	POD still relies on loose signatures/photos

Table 2.4.1 Comparison of the most well-known POD system in the current market

The table above is constructed to show the most well-known POD systems in the current market, as well as its respective Features, Strengths, and Weakness.

By analyzing the different systems above, there is a pattern found in each system. All major POD software relies on digital signatures, phone contact, and barcodes. According to the

feedback of these software, transparency via notifications, or live tracking is a major selling point, since customers want reassurance. Moreover, streamline the process with automation technology is a big area to be explored. For example, features such as instant POD documents sharing, or implementing **report generation** (audit logs) for POD records saves courier time and enhances efficiency.

Routific has the strengths in providing strong transparency and driver efficiency. It utilises presenting notifications and live tracking for customers to be in control of their parcel delivery status. According to the table above, several Software such as OptimoRoute integrate route planning with time windows to enhance the delivery efficiency.

However, several limitations and weaknesses have been found in these POD systems. One of which is universal, and considered more critical is that the photos taken within these systems are not well-structured. As this paper stated above, once the geolocation metadata is extracted, it needs to be converted into readable format for further processing. Since this metadata is embedded within the photo, the system is able to retrieve the location and time taken without an issue. However, if sent to the customers, unstructured photos with vaguely defined rules tend to have a negative impact on the consumer's experience. For example, a blurry photo taken by the courier, or just a black box of parcel laying on the ground, without location context and visual cues of the background. This has been a big issue across all Proof of Delivery systems.

There is also a limitation on the accuracy of the GPS data extracted from the image. Cameras within smartphones nowadays tend to capture and store the EXIF location data with up to 4 decimal accuracy, that is around 11 meters, when scaled to Real-World-Mapping.

The goal here is to extract **GPS location** and **date-time** information embedded in images. EXIF metadata ensures that **when** and **where** a photo was taken can be verified automatically. This prevents manual errors and lays the foundation for proof of delivery verification, such as the location and time of a parcel, if stated delivered.

However, with the retrieval of Geolocation Metadata embedded within an image, it sparks a new problem. Which is: How Accurate Does this Retrieved Geolocation Data Is?

Below is the Geodetic Conversion (also called the Coordinate Conversion) of raw GPS data stored as Degree, Minutes, Seconds (DMS) extracted from EXIF into Decimal Degrees (DD) format, which is the standard format for GPS coordinates.

How conversion works:

- EXIF stores coordinates as tuples of rational numbers:

```
(degrees, minutes, seconds) = (numerator, denominator)
```

- The conversion formula:

```
java
Decimal Degrees = Degrees + (Minutes / 60) + (Seconds / 3600)
```

- The process involves:
 - Extract degrees → `[0][0] / [0][1]`
 - Extract minutes → `[1][0] / ([1][1] * 60)`
 - Extract seconds → `[2][0] / ([2][1] * 3600)`

Directional Reference Check:

- EXIF also stores direction as N/S/E/W.
- If latitude = South (S) → make it negative.
- If longitude = West (W) → make it negative.

Example:

```
nginx
if lat_ref == 'S': latitude = -latitude
if lon_ref == 'W': longitude = -longitude
```

Output:

- Converted **Decimal Degree** coordinates → directly usable in Google Maps or any GIS system.
- Example: `3.1421° N, 101.6823° E`
- Enables:
 - Map plotting
 - Linking to delivery records
 - Accurate geotagging for verification

Figure 2.4.2, 2.4.3 Geodetic Conversion of raw GPS data into Decimal Degrees

According to the StackExchange forums, *Accuracy* is the tendency of your measurements to agree with the true values whereby *Precision* is the degree to which your measurements pin down an actual value.

As a principle, using too much precision is misleading and tends to result in inaccurate results. According to the International Yard & Pound Treaty of 1959 [18], ten million meters will take someone from the equator to a pole, which states 90 degrees.

- Therefore, 1 degree of Latitude covers about $10^7/90 = 111,111$ meters.
- 1 degree of Longitude is the same or less in length than a degree of Latitude, because the circles of latitude shrink down to the earth's axis as we move from the equator towards either pole.

Stated by Whuber, the sixth decimal place in one decimal degree has $111,111/10^6 = 1/9$ meter, which is about 4 inches of precision. Below is a table from Don Meltz, that shows Latitude coordination precision by the actual cartographic scale they purport:

Decimal Places	Aprox. Distance	Say What?
1	10 kilometers	6.2 miles
2	1 kilometer	0.62 miles
3	100 meters	About 328 feet
4	10 meters	About 33 feet
5	1 meter	About 3 feet
6	10 centimeters	About 4 inches
7	1.0 centimeter	About 1/2 an inch
8	1.0 millimeter	The width of paperclip wire.
9	0.1 millimeter	The width of a strand of hair.
10	10 microns	A speck of pollen.
11	1.0 micron	A piece of cigarette smoke.
12	0.1 micron	You're doing virus-level mapping at this point.
13	10 nanometers	Does it matter how big this is?
14	1.0 nanometer	Your fingernail grows about this far in one sec
15	0.1 nanometer	An atom. An atom! What are you mapping?

Figure 2.4.3 Latitude coordination precision by the actual cartographic scale purported

According to Abhi Beckert, a typical consumer GPS has on the order of 11 meters accuracy (or 4 decimal places) under good conditions (clear sky, no interference)[\[21\]](#). Therefore, a device might report 6 or 7 decimals, but only the first 4 decimals are reliable to within 11 meters.

Differential/GNSS augmentation can improve accuracy to sub-meter or centimeter levels [\[18\]](#), but that is rare in everyday photography and is bounded by hardware limitations. It is simply not feasible for a Proof of Delivery System. The key point is that **too many digits are misleading** if the measurement error is larger.

As one answer summarizes: The sixth decimal (11 cm) is more than good enough for tracking rivers, and over 10 decimals is overkill. *Worst-case error*: Real geotags can be way off. A forensic study found smartphone image locations sometimes off by 5–27 km. Even short of

that, dozens of meters of error are not uncommon. This is the limitation of a typical Geolocation Pin-point approach using GPS and EXIF method.

Lastly, the introduction of Cloud Storage is a major component that makes a POD system more robust in storing its information. However, EXIF data tend to be modifiable and change easily, challenging the data privacy of an individual. With research, there are very few integration of digital security methods that are processed on EXIF data, specifically the real-time location. This could result in serious consequences where data breaching happens, and making customer's rights to data privacy at risk.

CHAPTER 3 System Methodology

3.1 General Development Methodologies

The figure below shows the Iterative and Incremental development methodology lifecycle.

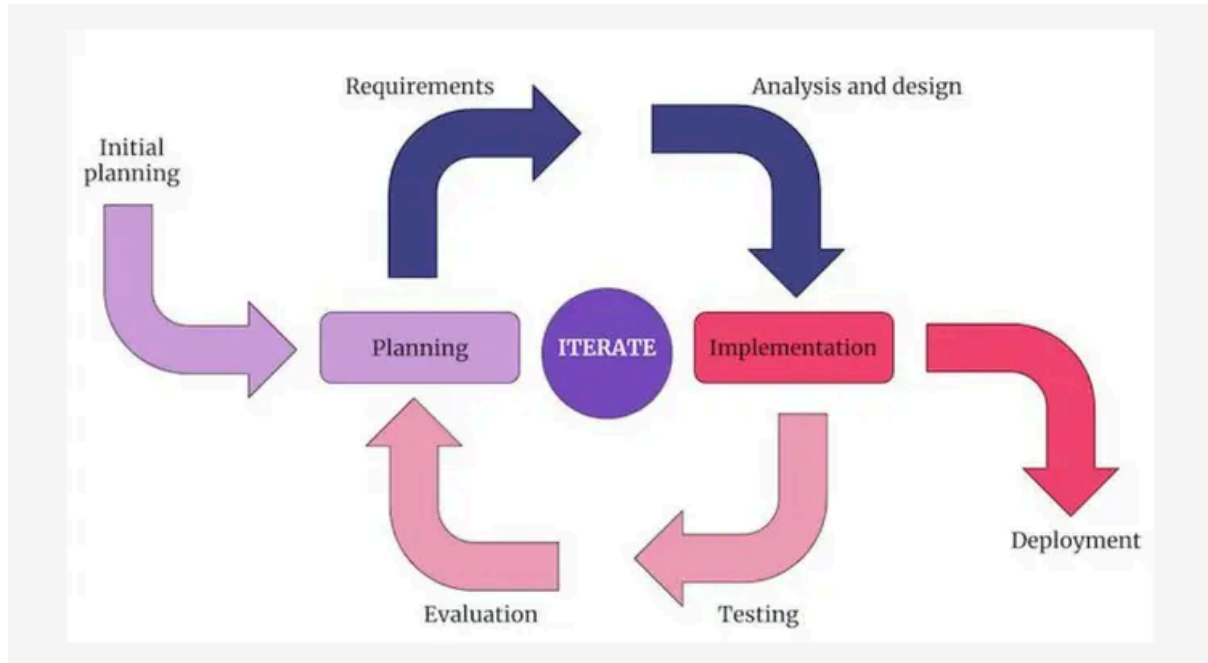


Figure 3.1.1 Iterative and Incremental Development Methodology Lifecycle in Software Systems

The development of this system follows an Iterative and Incremental Development approach. Developing a complex deep learning models for both Building Detection and Siamese Model Calculations that is needed to be integrated alongside a mobile application that works on EXIF Metadata carries significant computational and infrastructural challenges, particularly concerning the deployment and real-time inference on edge devices or cloud-based platforms [\[22\]](#).

Unlike traditional linear methodology (e.g., Waterfall Development Methodology) the approach allows for the project to be broken down into smaller, manageable cycles, known as Iterations. Each iteration focuses exclusively on achieving a portion of the final system functionality. This methodology is particularly advantageous in facilitating continuous

refinement and early error detection, which is crucial for systems reliant on precise metadata processing and computer vision algorithms [\[23\]](#).

Therefore, with an iterative and incremental methodology this project is divided into 3 distinct phases to ensure both core stability and advanced feature integration. Namely:

- Phase 1: Minimum Viable Product (MVP) of the Mobile Edge Device Application and Core Architecture
- Phase 2: Advanced Feature Development and Integration of Computer Vision Modules
- Phase 3: Operational Functions, Field Deployment, and Iterative Refinement

After the complete development and integration of the 3 phases, the system will be verified and field tested, in collaboration with GDEX to assess the system's performance under load and varying network conditions. The details and outcomes of the field testing is listed out in Chapter 6: System Evaluation.

3.1.1 Phase 1: The Minimum Viable Product (MVP)

The primary objective of this first layer of development is to establish a foundational software pipeline and data infrastructure. An MVP is defined as a version of a new product that allows a team to collect the maximum amount of validated learning with the least effort [\[24\]](#), [\[25\]](#).

- **Cross-Platform Foundation:**
 - The mobile application is developed using **React Native** and **Expo Application Services**, providing a robust operational foundation for cross-platform deployment.
- **Structured Capture Guidelines:**
 - The application has integrated a structured rule of capturing photos, such that for each image, it should enforce a dual-layer user interface composition. The application overlays building and parcel framing guides on the camera view to prevent users from capturing photos that are blurry, or lack significant context, such as the background location or a physical view of the parcel. This ensures all proof images meet a standardized visual format.
- **Geolocation Gating:**

- Implementation of a Geolocation Gating algorithm is presented in this phase. This verification logic handles GPS signal variance. The scope includes developing the mechanism to prevent photo proofs to be captured when location accuracy is poor and providing a time-bounded fallback mechanism for GPS-denied environments. The application should provide intuitive feedback to the users according to the quality and accuracy of this two-factor verification parameter.
- **Local-First Metadata Injection:**
 - The system utilizes a local processing engine to extract real-time telemetry, specifically the GPS coordinates and timestamps from the edge device and injects this data directly into the captured POD image's **EXIF header tags** before proceeding with the image to storage. This client-side processing ensures zero latency during the capture process to ensure delivery operations are not impeded by metadata injection procedures.
- **Cloud Infrastructure:**
 - For storage and backend routing, the system integrates **Supabase** (a cloud database and Backend As A Service Platform) to ensure secure storage of raw image proofs and modular scalability.

Key Deliverables: A functional mobile application with dual-layer HUD capture, a geolocation gating algorithm, EXIF metadata injection capabilities, and successful cloud synchronization with Supabase.

3.1.2 Phase 2: Advanced Feature Development and Integration of Computer Vision Modules

This phase evolves the MVP into a "Zero-Trust" Proof of Delivery pipeline by integrating edge-native deep learning models, transitioning the system from manual data collection to autonomous visual verification. This progression necessitates the development and training of specialized computer vision architectures, specifically for Barcode Recognition, Semantic Building Detection, and Siamese Neural Network similarity scoring.

To effectively train and validate these models, a proprietary empirical dataset was curated via in-situ field photography across diverse residential topologies in Kampar, Perak. An example of datasets is as shown in Figure 3.1.2 below:

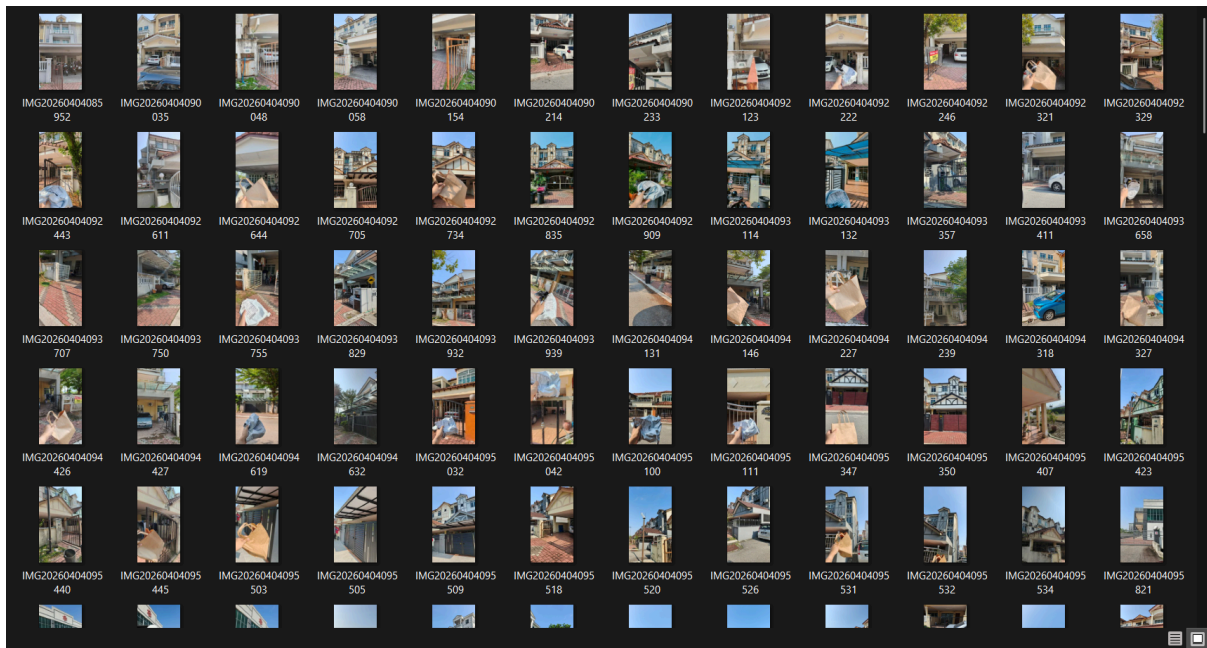


Figure 3.1.2 Terrace Housings Datasets collected for the Building Recognition Module

This localized data acquisition strategy was explicitly designed to capture real-world environmental variances (e.g., dynamic illumination, shadows) and the architectural homogeneity typical of the target deployment domain, ensuring the resulting models are highly robust in active logistics scenarios.

Once fine-tuned, the neural network weights were quantized into an edge-deployable format (e.g., **.tflite**) and seamlessly embedded directly into the existing React Native mobile framework introduced in Phase 1.

- **Edge AI Deployment:**

- The system deploys quantized **.tflite** models, specifically **MobileNetV3**, directly into the React Native frame processor. MobileNetV3 is optimized for mobile environments, offering significantly lower latency and reduced model size (often under 5MB) while maintaining high precision for real-time semantic validation [26], [27].

- **Automated Parcel Identification:**

- AI-driven Optical Character Recognition (OCR) for barcode scanning from the Consignment Notes attached to the physical parcels are integrated to extract Unique Identifiers directly from the live camera feed.

- **Edge-Native Convolutional Neural Network for Semantic Validation:**
 - This module deploys a lightweight Convolutional Neural Network (CNN) to perform real-time semantic validation of the delivery environment. To achieve low-latency inference on hardware-constrained mobile devices, the model is quantized and deployed as an edge-native asset within the application framework. The network continuously processes the live camera feed, extracting structural features to calculate a probabilistic confidence score regarding the presence of architectural elements. This score is evaluated asynchronously and rendered on the client interface, serving as an algorithmic gatekeeper that actively prevents the capture sequence until a valid structural target is verified. Note that as mentioned in Section 1.4.1 System Boundaries, this validation feature serves as a necessary prerequisite foundational infrastructure for fully autonomous logistics integration in the future. Currently the workflow is implemented via an independent external web application instead of directly into the client edge device.
- **Cross-Temporal Similarity Verification via Siamese Neural Networks:**
 - To authenticate delivery locations across varying temporal and environmental conditions, the system implements a deep Siamese Neural Network. This architecture utilizes twin convolutional sub-networks with shared topological weights to project paired input images, a newly captured Proof of Delivery (POD) image and a historical baseline reference into a unified, high-dimensional latent feature space. By mapping semantically similar architectural structures in close proximity within this latent space, the network computes a definitive mathematical Euclidean distance via Cosine Similarity between the two feature vectors. This derived distance translates into a robust similarity score, enabling objective, automated verification for delivery operations [\[5\]](#), [\[28\]](#).
- **Key Deliverables:** Implementation of autonomous visual verification, including barcode/OCR recognition, building recognition scoring, and Siamese Network-based similarity analysis.

3.1.2 Phase 3: Operational Functions, Field Deployment, and Iterative Refinement

The final phase focuses on completing the operational pipeline for real-world logistics environments and conducting rigorous stress testing. This includes the development of a POD Retrieval Landing Page sent by Couriers to Recipients. And development of tools and system used for evaluating the performance and robustness of the deployed computer vision models under diverse environmental conditions [\[29\]](#).

- **Custom Metadata-Preserving POD Retrieval Portal:**
 - Standard third-party (e.g., WhatsApp, Messengers) communication channels typically utilize aggressive image compression algorithms that inherently strip embedded EXIF metadata during transmission. To mitigate this data degradation, a dedicated web-based retrieval portal was engineered. This architecture ensures that the recipient can access the Proof of Delivery (POD) imagery while preserving the complete integrity of the injected spatial and temporal data. The portal renders a comprehensive verification dashboard, exposing granular delivery telemetry including the exact temporal timestamp, raw GPS coordinates, interactive geospatial mapping integration, the authenticated POD image, and a lossless download vector for dispute resolution.
- **Dynamic Access Control:**
 - After a successful delivery attempt, the delivery record will be uploaded to Supabase. Following a verified autonomous delivery event and the successful commitment of the data payload to the centralized backend ledger, a dynamic Universal Resource Locator (URL) will be generated and attached along with the parcel UID and sent to the recipients after a successful delivery attempt. This unique endpoint acts as a secure, read-only access key to the specific delivery record. The application subsequently leverages native operating system share-sheet protocols to route this secure portal link directly to the recipient. This enables seamless notification via external messaging architectures while entirely bypassing the metadata-stripping constraints of direct image sharing.
- **Empirical In-Situ Validation (Field Testing in Collaboration with GDEX):**

- To validate the architecture under real-world operational constraints, the system was deployed in-situ alongside active GDEX logistics fleets. This collaborative field-testing phase was explicitly designed to stress-test the deep learning models against adversarial edge cases not easily replicated in controlled environments. Specifically, the field deployment captured critical telemetry regarding severe environmental variance (e.g., Dynamic Illumination Decay, Extreme Glare, Network Dead-zones, low-resolutions of Consignment Notes) and evaluated the system's robustness when processing complex topological environments, such as architecturally homogeneous residential grid layouts (e.g., Guard Houses, Shop-lots, etc.)
- **Telemetric System Optimization and Calibration:**
 - The final methodological phase involved the empirical calibration of the system's operational hyperparameters using the telemetry data harvested during live deployment. This optimization process focused on iteratively fine-tuning the probabilistic confidence thresholds of both the CNN and Siamese networks. The primary objective was to achieve an optimal equilibrium between classification accuracy and computational overhead, specifically minimizing edge-level inference latency (`inference_time_ms`) to guarantee seamless, zero-friction execution for the logistics end-user without compromising the mathematical integrity of the zero-trust pipeline.
- **Key Deliverables:**
 - A complete end-to-end operational pipeline with metadata-preserving retrieval, validated through field deployment and real-world performance tuning.

3.2 System Architecture Diagram

The overall system developed consists of 3 layers:

- **LAYER 1 - Edge Layer (Driver's Mobile Device)** : Color Coded with Blue in figure 3.2.1
- **LAYER 2 - Cloud Layer (Supabase BaaS)** : Color Coded with Orange in figure 3.2.1
- **LAYER 3 - Presentation Layer (Web)** : Color Coded with Green in figure 3.2.1

The figure below is a system design diagram to showcase the entire workflow of the system, and how each part interacts with each other.

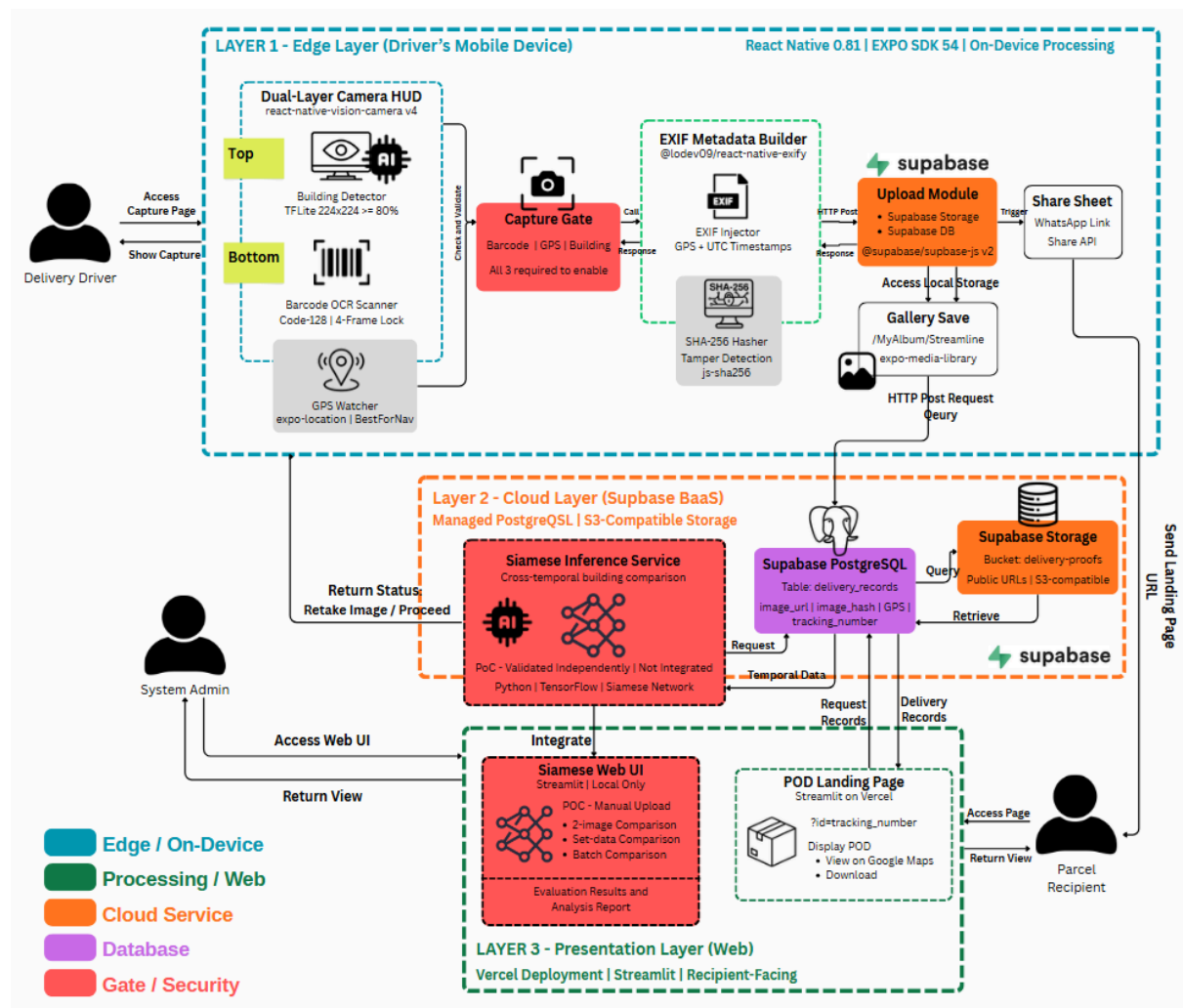


Figure 3.2.1 System Architecture Diagram

This modular, three-layered architecture facilitates scalability and maintainability, allowing for independent optimization and upgrades of each component without disrupting the entire system [\[30\]](#).

Three external actors bookend the system:

- **Driver**, who initiates data capture at the Edge Layer
- **Parcel Recipient**, who consumes the verified output at the Presentation Layer
- **System Admin**, who access the Siamese Web UI for manual uploading of POD images for comparisons.

3.2.1 Layer 1: Edge Layer (Driver's Mobile Device)

Definition: This layer encompasses the on-device processing units responsible for initial data acquisition and preliminary computational tasks, such as real-time image capture, Building Detection ML model, and embedded EXIF metadata extraction [\[31\]](#), [\[32\]](#).

Component Breakdown.

- The Edge Layer is implemented in React Native 0.81 with Expo SDK 54 and comprises nine components. The Driver connects to a Dual-Layer Camera HUD powered by react-native-vision-camera v4, featuring a Building Detector (top HUD frame processor) and Barcode Scanner (bottom HUD via useCodeScanner). A GPS Watcher (expo-location, BestForNavigation accuracy) operates parallel to the HUD group. These inputs converge at a Capture Gate—a security checkpoint enforcing conjunctive validation prior to image capture.
- Post-capture, the workflow proceeds to the EXIF Injector (@lodev09/react-native-exify), embedding GPS coordinates and UTC timestamps into the JPEG. The image advances to the SHA-256 Hasher (js-sha256) for tamper-evident fingerprinting. A secondary branch from the EXIF Injector enables local persistence via Gallery Save (expo-media-library). The primary path reaches the Upload Module (@supabase/supabase-js v2), which initiates parallel operations: JPEG upload to Supabase Storage and record insertion into Supabase PostgreSQL, bridging to Layer 2. The module also activates a Share Sheet to deliver a secure WhatsApp link directly to the Recipient.

3.2.2 Layer 2: Cloud Layer (Supabase BaaS)

Definition: This layer comprises the remote server infrastructure that handles more intensive computational demands, including advanced image processing, deep learning model inference, and secure data storage [33].

- **Component Breakdown:**

The layer is annotated as Managed PostgreSQL, S3-Compatible Storage, and Deno Edge Functions. Two primary components receive data from Layer 1: **Supabase Storage** (bucket: **delivery-proofs**, S3-compatible, public URLs) receives the uploaded JPEG, and **Supabase PostgreSQL** (table: **delivery_records**, storing **image_url**, **image_hash**, GPS coordinates, and **tracking_number**) receives the structured metadata record. A solid arrow descends from PostgreSQL to the POD Landing Page in Layer 3 when requested to be viewed, it past down the requested parcel UID as parameters.

- Below the primary row, a **Siamese Inference Service** (Python, TensorFlow, Siamese Network) appears with a dashed-rose border denoting its proof-of-concept status. An arrow labelled "Temporal Data" descends from this service into the Siamese Web UI component described in Layer 3.

3.2.3 Layer 3: Presentation Layer (Web)

Definition: This layer focuses on the user-facing interfaces, providing intuitive visualization of delivery data, interactive mapping, and tools for dispute resolution and administrative oversight, as well as the dynamic access control for recipients.

Component Breakdown:

- The layer is annotated as Vercel Deployment, Streamlit, and Recipient-Facing. The **Recipient** actor sits at the left of the layer, connected via an arrow to the **POD Landing Page** (Streamlit on Vercel), which accepts a **?id=tracking_number** parameter to display the corresponding proof-of-delivery record. This landing page is the destination of the WhatsApp link originating from the Share Sheet in Layer 1.

- A second component, the **Siamese Web UI** (Streamlit, Local Only), is rendered indicating its proof-of-concept status as a manual two-image comparison tool, not connected to the production pipeline.

3.3 Use-Case Diagram and Use-Case Descriptions

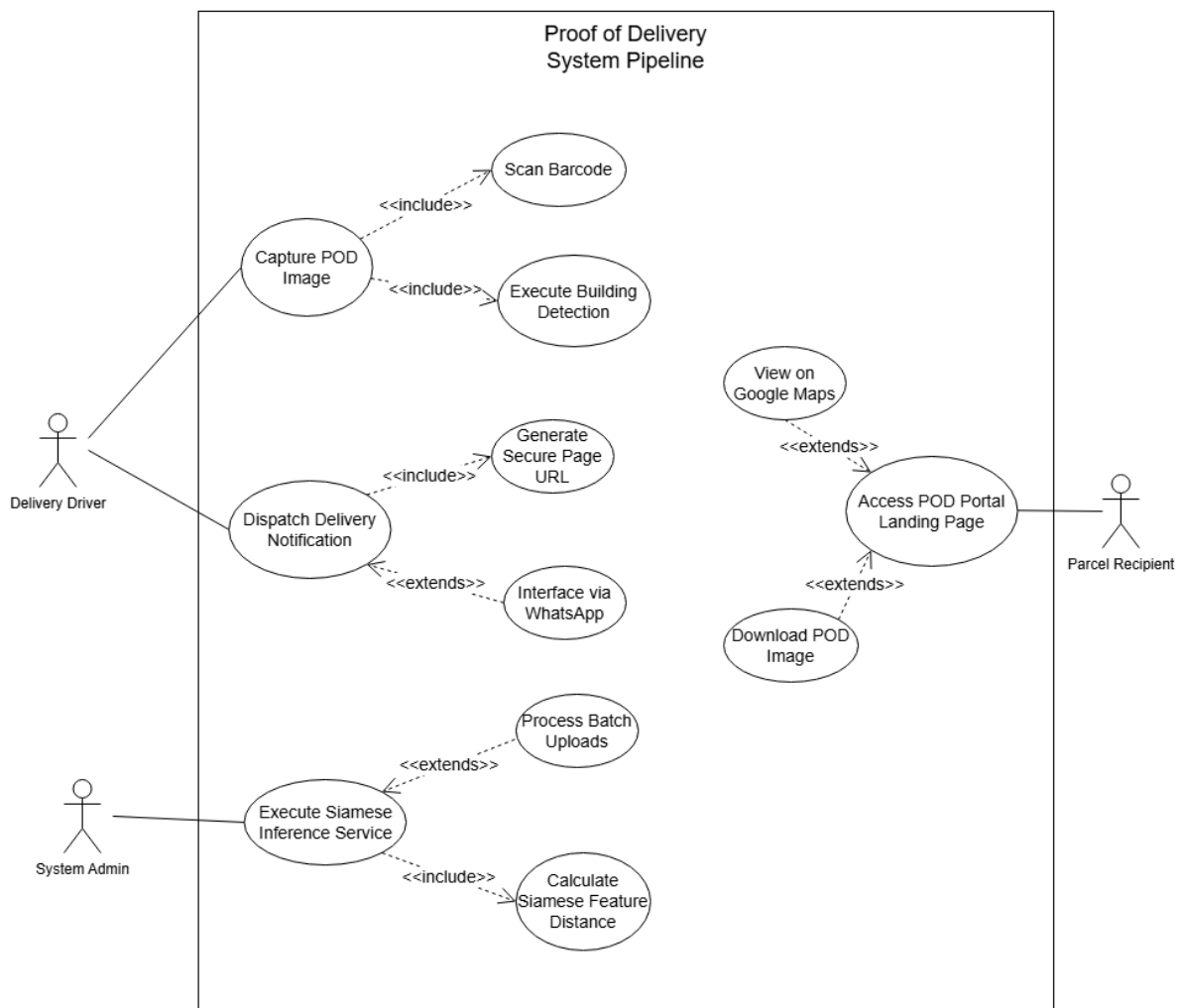


Figure 3.3.1 Use-Case Diagram depicting actors and interactions

The diagram in Figure 3.2.2 delineates the operational scope for each actor, including the Delivery Driver, Parcel Recipients, and System Admin by mapping their respective interactions with the system's core pipeline of Capturing and Delivery POD Images. It outlines the functionalities (use cases) that the system provides and how these actors utilize them.

3.3.1 Use-Case Descriptions

There are a total of 4 use-case descriptions listed out in this section. Each use case is further elaborated with detailed steps, pre-conditions, and post-conditions to provide a comprehensive understanding of the system's operational flow and actor interactions.

Use Case ID	Use Case Name	Primary Actor	Purpose
UC01	Capture POD Image	Delivery Driver	To capture proof of delivery image with barcode scanning and building detection for delivery verification
UC02	Execute Siamese Inference Service	System (automated process)	To verify delivery location authenticity by comparing captured POD image with reference building images using Siamese neural network
UC03	Access POD Portal Landing Page	Parcel Recipient	To allow parcel recipients to access their delivery proof through a secure web portal
UC04	Dispatch Delivery Notification	Delivery Driver, System	To notify parcel recipients of successful delivery with secure access to POD via WhatsApp or other channels

Table 3.3.1: The overall view of Use Case Descriptions listed out in this section, including their ID, Name, Primary Actor, and Purpose

UC01 Capture POD Image:

Field	Value	
ID	UC01	
Purpose	To capture proof of delivery image with barcode scanning and building detection for delivery verification	
Primary Actor(s)	Delivery Driver	
Trigger	Delivery driver arrives at delivery location and initiates POD image capture	
Sub-Functions	• Scan Barcode • Execute Building Detection • Capture Image • Validate Image Quality	
Pre-conditions	• System is active and accessible • Driver is logged in • GPS/camera permissions granted • Delivery location is reached	
Scenario Name	Step	Action
Main Flow	1	Driver navigates to delivery address
	2	Driver launches POD image capture function
	3	System activates camera interface
	4	System scans barcode on parcel/delivery note
	5	System executes building detection to verify location
	6	Driver captures POD image
	7	System validates image quality and completeness
	8	System saves POD image with metadata
Alternate Flow 1 - Barcode Scan Failure	4.1	System fails to scan barcode
	4.2	System prompts driver to manually enter tracking number
	4.3	Driver enters tracking number manually
	4.4	System validates tracking number
	4.5	Resume from Step 5

Alternate Flow 2 - Building Detection Failure	5.1	Building detection fails to match expected location
	5.2	System flags location mismatch warning
	5.3	Driver confirms correct location or adjusts GPS
	5.4	Resume from Step 6
Alternate Flow 3 - Poor Image Quality	7.1	System detects poor image quality (blur, low light)
	7.2	System prompts driver to retake image
	7.3	Resume from Step 6
Post-conditions	-	POD image is captured, validated, and stored with associated barcode and location data

Table 3.3.2 Use-case description for UC01 Capture POD Image

UC02 Execute Siamese Inference Service:

Field	Value	
ID	UC02	
Purpose	To verify delivery location authenticity by comparing captured POD image with reference building images using Siamese neural network	
Primary Actor(s)	System (automated process)	
Trigger	POD image is successfully captured and validated	
Sub-Functions	• Calculate Siamese Feature Distance • Compare with Reference Images • Process Batch Uploads • Generate Similarity Score	
Pre-conditions	• POD image is captured and stored • Reference building images are available in database • Siamese model is trained and operational	
Scenario Name	Step	Action
Main Flow	1	System receives validated POD image
	2	System extracts features from POD image using Siamese network
	3	System retrieves reference building images for delivery address
	4	System calculates Siamese feature distance between POD and reference images
	5	System generates similarity score
	6	System validates if similarity score exceeds threshold
	7	System marks delivery location as verified
Alternate Flow 1 - Batch Upload Processing	1.1	Multiple POD images uploaded simultaneously
	1.2	System queues images for batch processing
	1.3	System processes each image in parallel
	1.4	Resume from Step 2 for each image

Alternate Flow 2 - Low Similarity Score	6.1	Similarity score below threshold
	6.2	System flags delivery for manual review
	6.3	System notifies supervisor of location mismatch
	6.4	Supervisor reviews and approves/rejects delivery
Alternate Flow 3 - No Reference Image Available	3.1	No reference images found for delivery address
	3.2	System marks delivery for manual verification
	3.3	System stores POD image as new reference (pending approval)
Post-conditions	-	Delivery location is verified through Siamese inference, and similarity score is recorded in system

Table 3.3.3 Use-case description for UC02 Execute Siamese Inference Service

UC03 Access POD Portal Landing Page:

Field	Value	
ID	UC03	
Purpose	To allow parcel recipients to access their delivery proof through a secure web portal	
Primary Actor(s)	Parcel Recipient	
Trigger	Recipient receives delivery notification with secure URL link	
Sub-Functions	• View on Google Maps • Download POD Image • Verify Delivery Details • View Delivery Timestamp	
Pre-conditions	• Delivery is completed • POD image is uploaded • Secure URL is generated and sent to recipient	
Scenario Name	Step	Action
Main Flow	1	Recipient clicks on secure URL from notification
	2	System validates URL and authentication token
	3	System loads POD portal landing page
	4	System displays POD image, delivery details, and timestamp
	5	Recipient views delivery proof
Alternate Flow 1 - View Location on Google Maps	5.1	Recipient clicks "View on Google Maps" button
	5.2	System retrieves GPS coordinates from delivery metadata
	5.3	System opens Google Maps with delivery location marked
	5.4	Recipient views delivery location on map
Alternate Flow 2 - Download POD Image	5.1	Recipient clicks "Download POD Image" button
	5.2	System prepares POD image for download
	5.3	System initiates image download to recipient's device

	5.4	Recipient saves POD image locally
Alternate Flow 3 - Invalid/Expired URL	2.1	System detects invalid or expired authentication token
	2.2	System displays error message
	2.3	System provides option to request new access link
	2.4	End use case
Post-conditions	-	Recipient successfully views delivery proof and can download POD image or view location on map

Table 3.3.4 Use-case description for UC03 Access POD Portal Landing Page

UC04 Dispatch Delivery Notification:

Field	Value	
ID	UC04	
Purpose	To notify parcel recipients of successful delivery with secure access to POD via WhatsApp or other channels	
Primary Actor(s)	Delivery Driver, System	
Trigger	Delivery driver completes delivery and uploads POD image	
Sub-Functions	• Generate Secure Page URL • Send Notification via WhatsApp • Send SMS/Email Notification • Log Notification Status	
Pre-conditions	• Delivery is marked as complete • POD image is uploaded and verified • Recipient contact information is available	
Scenario Name	Step	Action
Main Flow	1	Driver marks delivery as complete in system
	2	System validates POD image upload
	3	System generates unique secure page URL with authentication token
	4	System retrieves recipient contact details
	5	System prepares notification message with secure URL
	6	System sends notification to recipient
	7	System logs notification dispatch status
Alternate Flow 1 - WhatsApp Notification	6.1	System detects WhatsApp as preferred communication channel
	6.2	System interfaces with WhatsApp Business API
	6.3	System sends WhatsApp message with secure URL and delivery details
	6.4	System receives delivery confirmation from WhatsApp
	6.5	Resume from Step 7

Alternate Flow 2 - SMS Fallback	6.1	WhatsApp delivery fails or not available
	6.2	System falls back to SMS notification
	6.3	System sends SMS with secure URL
	6.4	Resume from Step 7
Alternate Flow 3 - Email Notification	6.1	Recipient has no mobile number registered
	6.2	System sends email notification with secure URL
	6.3	Email includes POD image as attachment
	6.4	Resume from Step 7
Alternate Flow 4 - Notification Failure	6.1	All notification channels fail
	6.2	System logs notification failure
	6.3	System flags delivery for manual follow-up
	6.4	Customer service contacts recipient manually
Post-conditions	-	Recipient receives delivery notification with secure access to POD portal, and notification status is logged in system

Table 3.3.5 Use-case description for UC04 Dispatch delivery notifications

3.4 Activity Diagram:

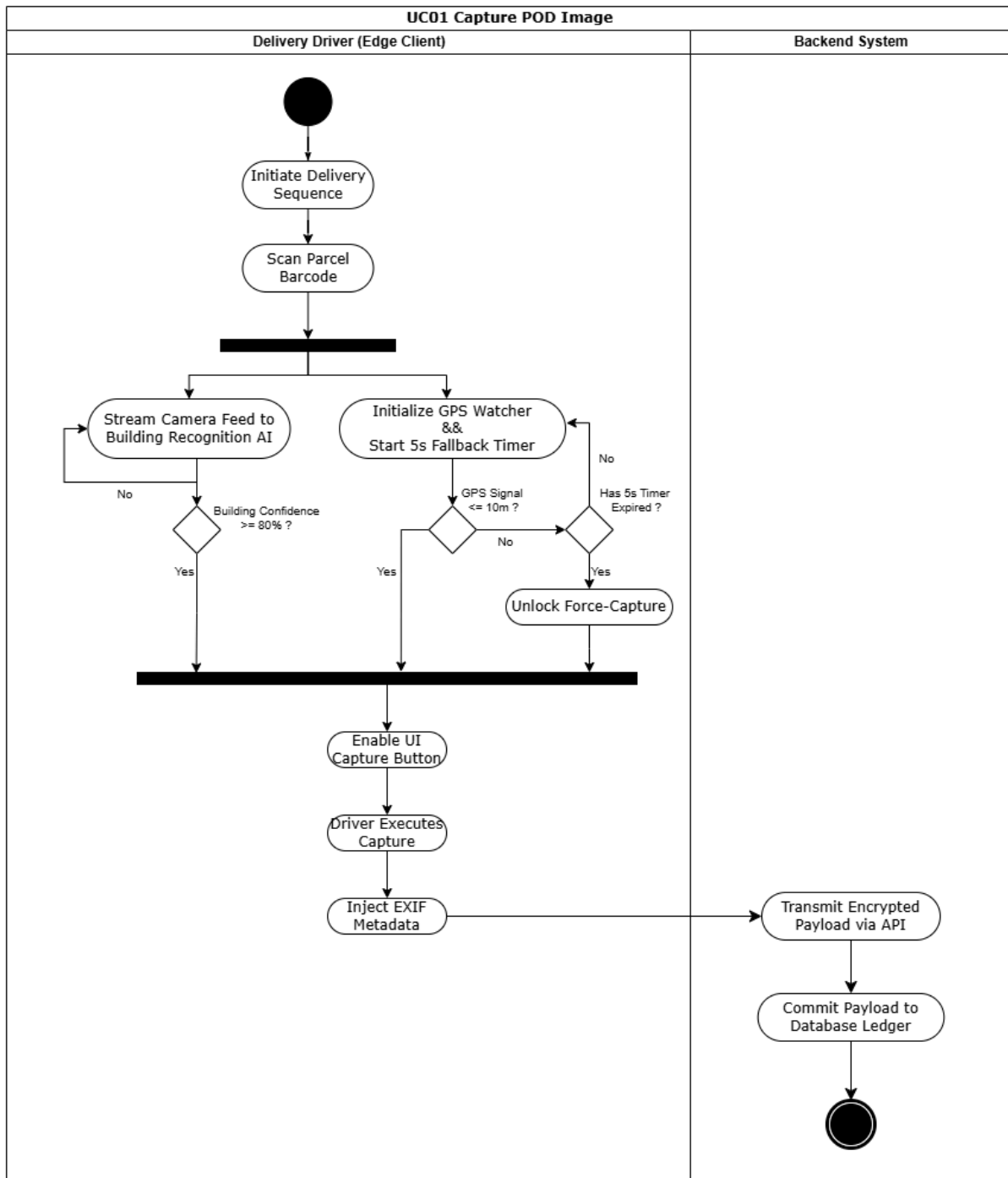


Figure 3.4.1 Activity Diagram for Edge-Native Data Collection

UC01: Capture POD Image

Purpose: Explains how the mobile app forces semantic validation and cryptographic EXIF anchoring *before* data ever reaches the cloud.

After Barcode capture:

- The GPS watcher continuously streams coordinates. The accuracy signal classifies each reading as green (≤ 10 m), yellow (≤ 50 m), or red (> 50 m / no fix). A 5-second fallback timer starts when the screen mounts; if the signal doesn't reach green in time, force-capture is unlocked so the user can still take a photo with lower accuracy. Reaching green at any point cancels the timer and reverts to normal verified capture.
- Because logistics drivers cannot afford to wait for asynchronous hardware to sequentially load, the system utilizes a concurrent architecture. Upon barcode verification, the semantic AI, the spatial GPS listener, and the temporal fallback timer are instantiated simultaneously. This creates a race-condition evaluation loop that guarantees either high-fidelity data capture, or operational continuity via the 5-second Force-Capture fallback.

Activity Diagram 2: The Decoupled Siamese Inference Pipeline

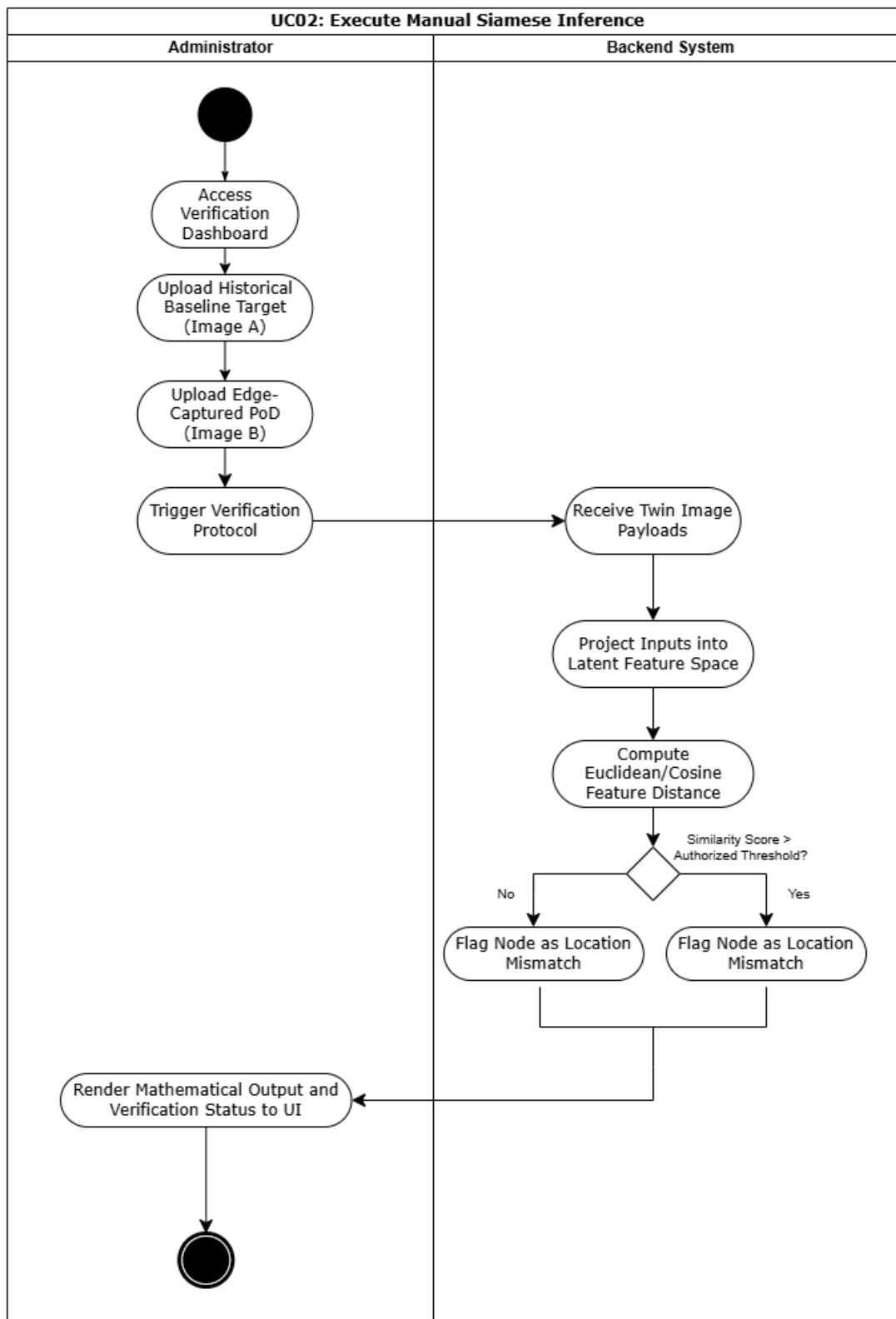


Figure 3.4.1 Activity Diagram for The Decoupled Siamese Inference Pipeline

UC02: Execute Manual Siamese Inference

Bachelor of Computer Science (Honours)

Faculty of Information and Communication Technology (Kampar Campus), UTAR

Purpose: Explains the standalone proof-of-concept AI module where the administrator tests the mathematical viability of the cross-temporal location verification.

Activity Diagram 3: Liability Transfer & Client Retrieval

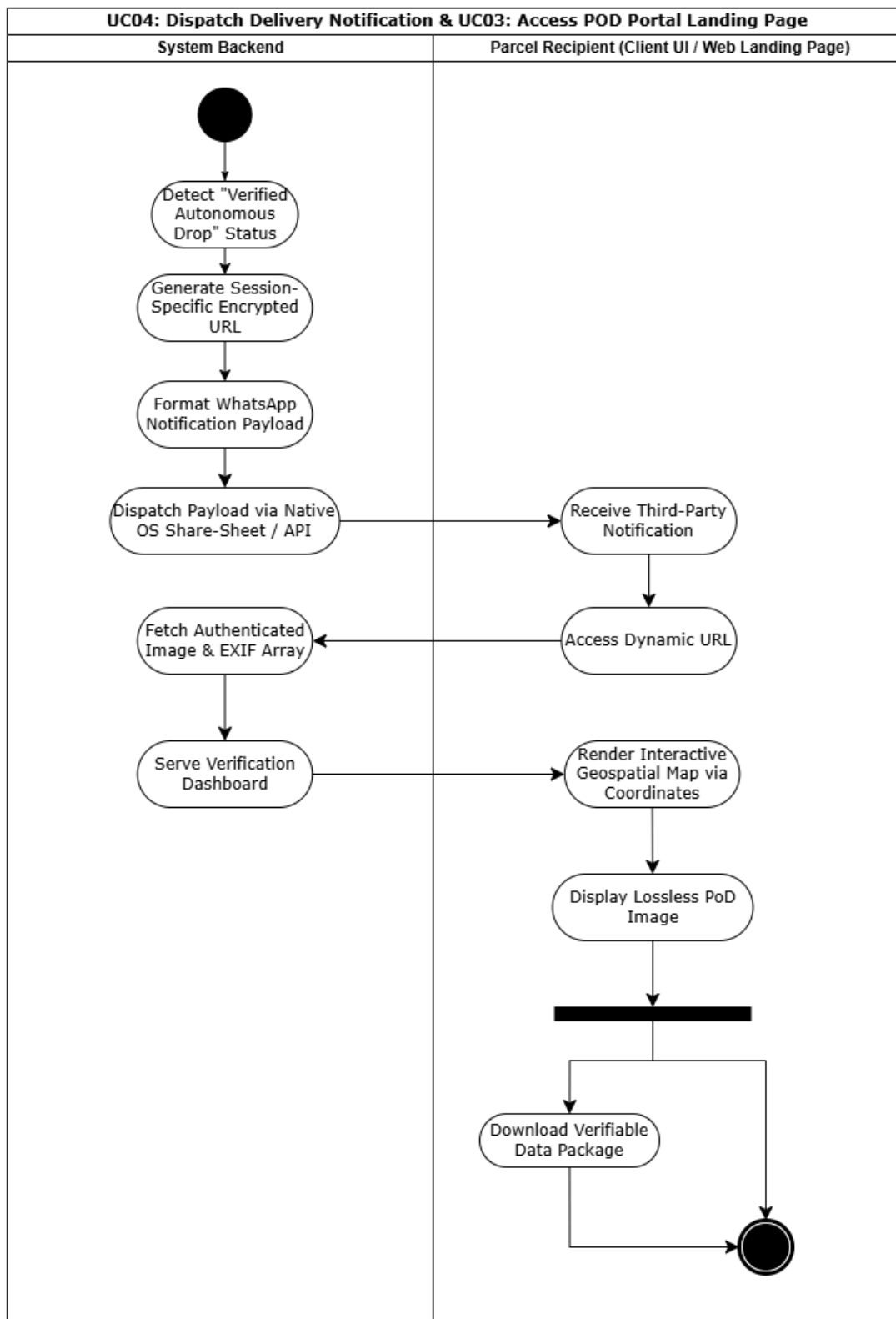


Figure 3.4.1 Activity Diagram for Liability Transfer & Client Retrieval

UC04: Dispatch Delivery Notification & UC03: Access POD Portal Landing Page

Purpose: Explains how the system autonomously bridges the gap between the authenticated data and the end-user, closing the loop on the unattended delivery.

CHAPTER 4 System Design

4.1 System Block Diagram

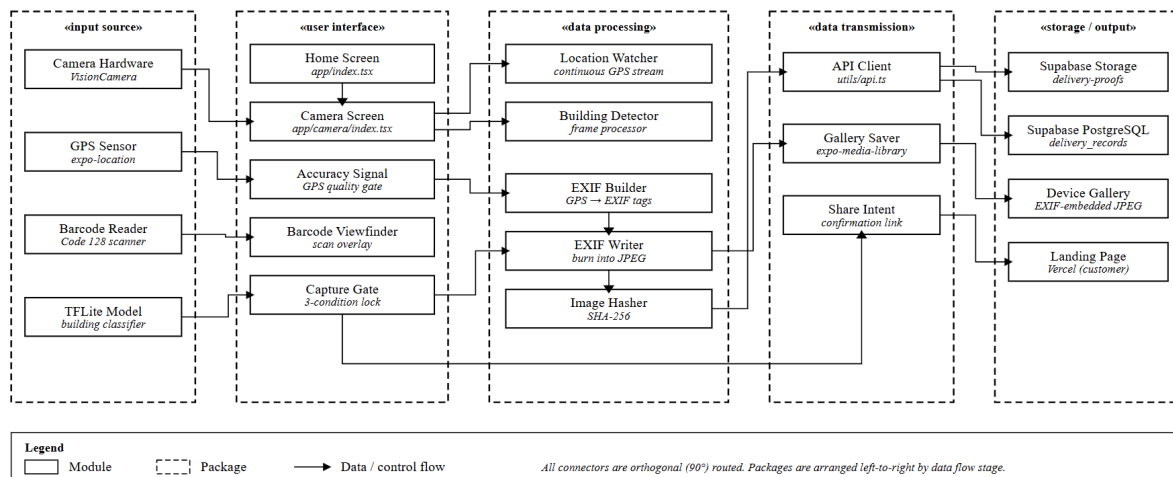


Figure 4.1.1 System Block Diagram

The system block diagram provides a simplified, abstract overview of the project's core functional groupings, decomposing the delivery verification process into its primary logical responsibilities. Unlike the architecture diagram which details physical deployment, this block diagram focuses on conceptual "chunks", namely the

- Input Sources
- User Interface
- Data Processing
- Data transmission
- Storage/Output

to establish the system's workflow and modular responsibilities before diving into technical implementation.

4.2 System Component Diagram

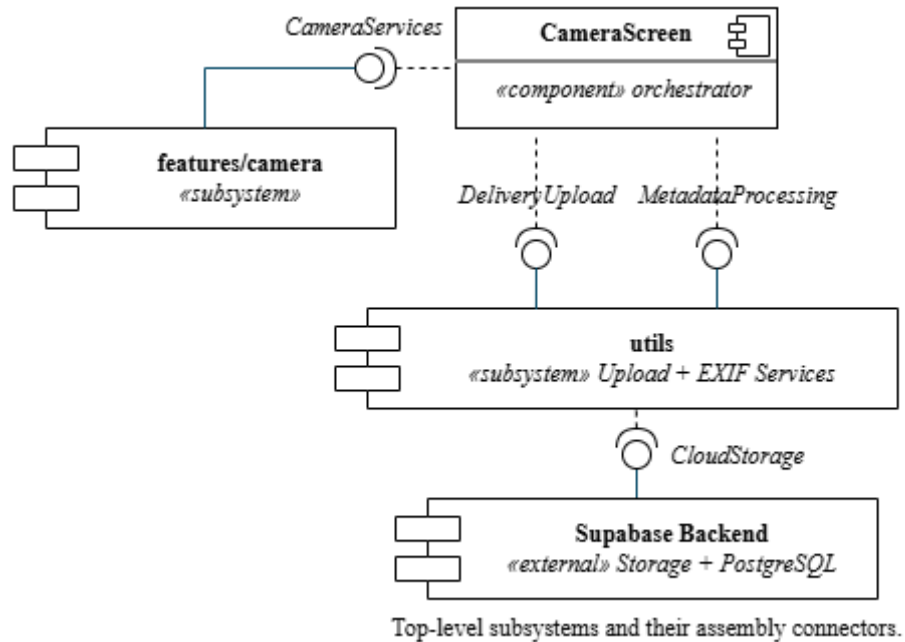


Figure 4.2.1 High Level Component Design Diagram

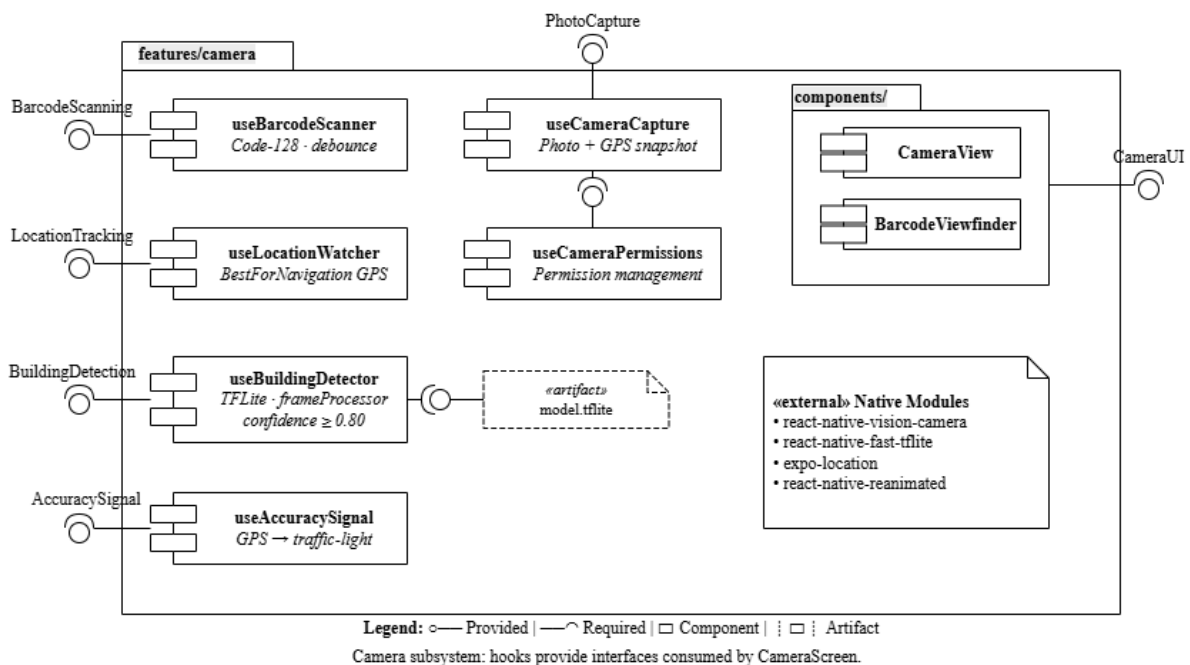


Figure 4.2.2 Camera Subsystem Component Design Diagram

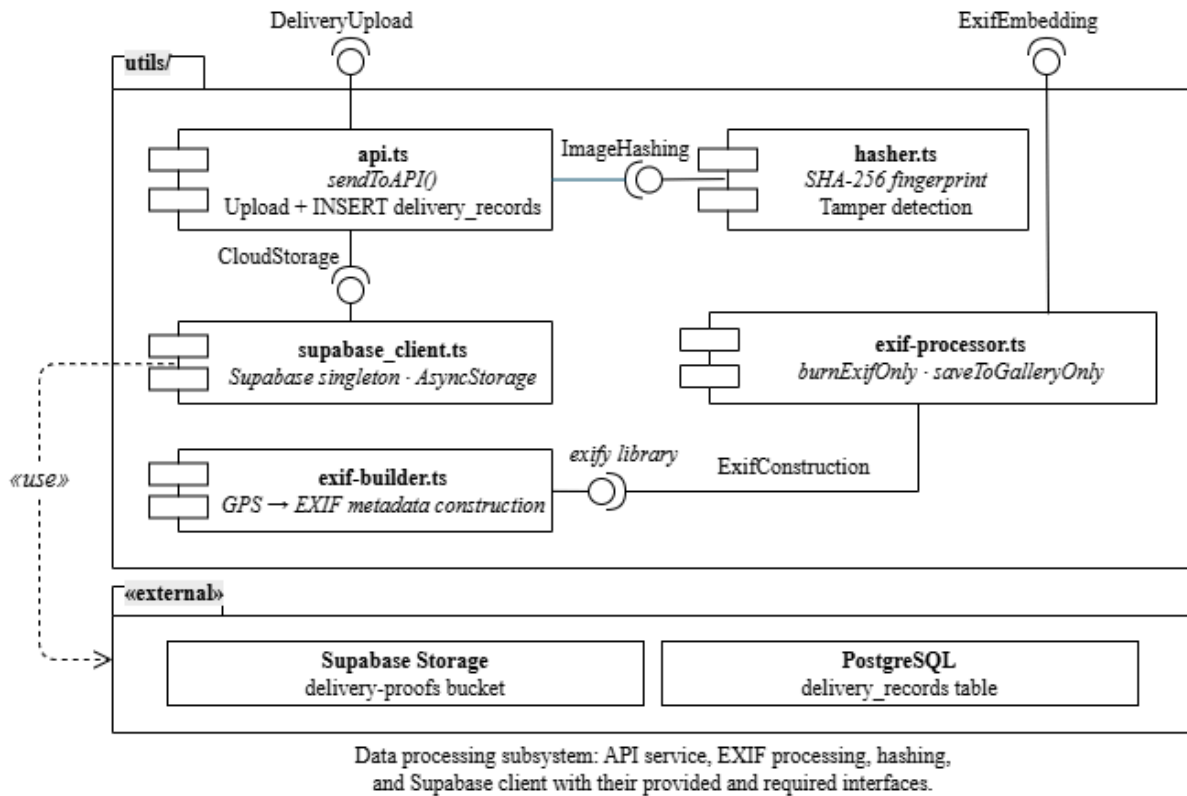


Figure 4.2.2 Data Processing Component Design Diagram

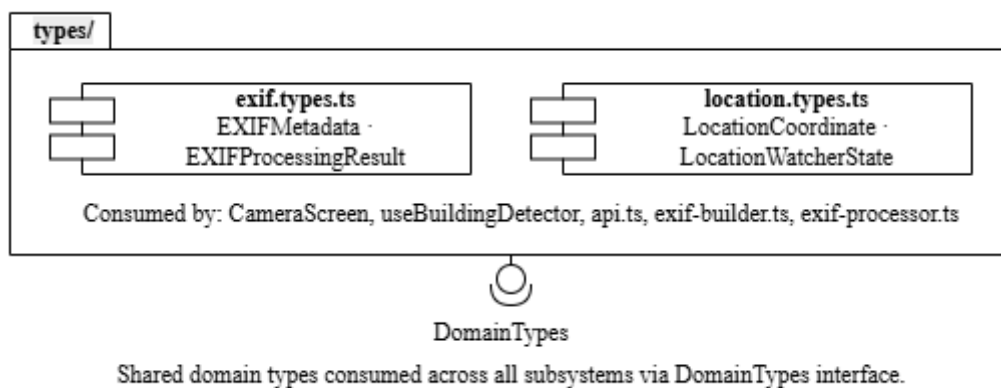


Figure 4.2.4 Shared Types Component Design Diagram

The system component diagram provides a formal UML representation of the software's internal architecture, focusing on the encapsulated, replaceable modules that form the delivery verification system. By detailing the provided and required interfaces between core modules, namely:

Figure 4.2.1	(a) Top-Level Overview	CameraScreen → features/camera, utils, Supabase Backend
Figure 4.2.2	(b) Camera Subsystem	6 hooks, components / package, model.tflite artifact, native modules note
Figure 4.2.3	(c) Data Processing	api.ts, hasher.ts, supabase_client.ts, exif-builder.ts, exif-processor.ts, backend
Figure 4.2.4	(d) Shared Types	exif.types.ts, location.types.ts, DomainTypes interface

These diagram illustrates the functional contracts and dependencies within the codebase. The modular decomposition ensures a clear understanding of how the React Native application's internal units interact to execute the system's computer vision and metadata extraction tasks.

4.3 Data and Database Design

The system persists delivery proof-of-delivery (POD) data in a **Supabase** backend comprising a PostgreSQL database, object storage buckets, and a serverless Edge Function. Two primary tables store delivery records (production and field-test), two storage buckets hold POD images, and a database webhook triggers SMS notifications via an Edge Function.

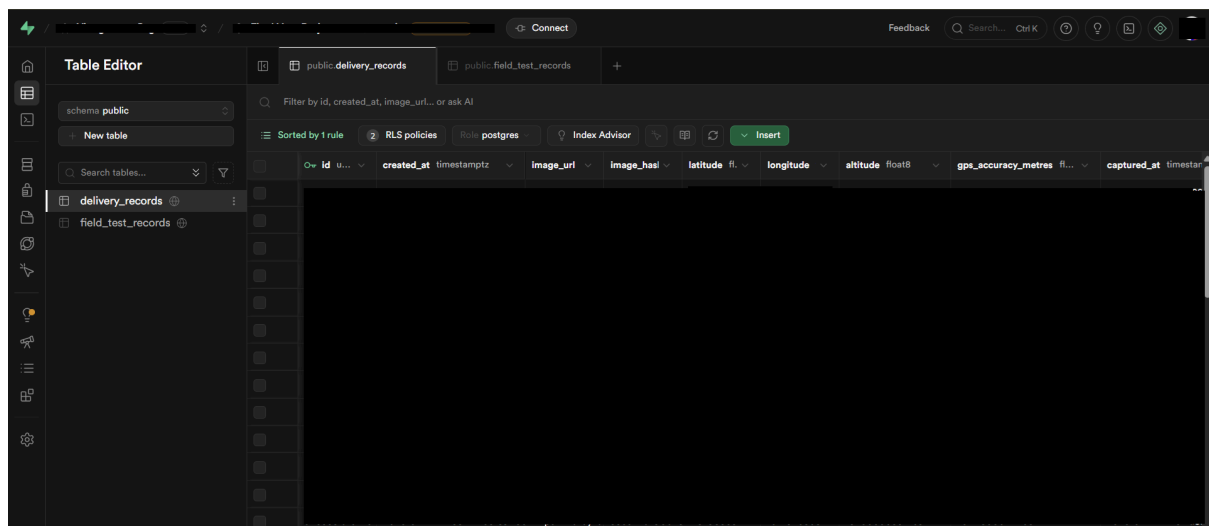


Figure 4.3.1 A Screenshot of the Supabase Table Dashboard with records redacted

4.3.1 Data Dictionary

A. EXIF Tags Captured

The system writes a focused subset of EXIF tags into every POD image before upload. The `buildExifFromCoordinate()` function (the detailed implementation is listed in **Chapter 4 Section 4.5.2**) produces the following tags from the device's live GPS fix:

EXIF Tag	EXIF IFD	Data Type	Example Value	Purpose
<code>DateTimeOriginal</code>	Exif IFD	ASCII string (20 chars)	<code>2025:04:29 16:05:33</code>	Exact local time the rider pressed the shutter
<code>DateTimeDigitized</code>	Exif IFD	ASCII string (20 chars)	<code>2025:04:29 16:05:33</code>	Time the image was stored as digital data
<code>DateTime</code>	IFD0	ASCII string (20 chars)	<code>2025:04:29 16:05:33</code>	Last-modified timestamp of the file
<code>GPSLatitude</code>	GPS IFD	RATIONAL (degrees)	<code>3.215180</code>	WGS-84 latitude of capture location
<code>GPSLongitude</code>	GPS IFD	RATIONAL (degrees)	<code>101.728430</code>	WGS-84 longitude of capture location
<code>GPSAltitude</code>	GPS IFD	RATIONAL (metres)	<code>45.2</code>	Altitude above/below sea level
<code>GPSAltitudeRef</code>	GPS IFD	BYTE (0 or 1)	<code>0</code>	0 = above sea level, 1 = below
<code>GPSTimeStamp</code>	GPS IFD	ASCII string (8 chars)	<code>08:05:33</code>	UTC time of GPS fix

Table 4.3.1 shows EXIFMetadata injected in each POD image. The full EXIFMetadata interface is attached in Appendix, it defines 30 optional tags spanning camera settings, exposure, image dimensions, and focal length. However, only the 9 tags above are **actively written** by the application; the remaining tags exist in the type definition to support future read-back or forensic analysis of third-party images.

B. 'delivery_records' Table — Column Dictionary

This is the production table inserted into by sendToAPI() in `utils/api.ts` (The original code is listed in APPENDIX)

Column Name	PostgreSQL Type	Nullable	Default	Constraint	Description
<code>id</code>	<code>uuid</code>	NO	<code>gen_random_uuid()</code>	PRIMARY KEY	Auto-generated unique row identifier
<code>created_at</code>	<code>timestamp tz</code>	NO	<code>now()</code>	—	Server-side insertion timestamp
<code>tracking_number</code>	<code>text</code>	NO	—	—	Parcel barcode / tracking number scanned by driver
<code>image_url</code>	<code>text</code>	NO	—	—	Public URL of POD image in Supabase Storage
<code>image_hash</code>	<code>text</code>	NO	—	—	SHA-256 hex digest of image file (tamper detection)
<code>latitude</code>	<code>float8</code>	NO	—	—	GPS latitude at capture (decimal degrees, WGS-84)
<code>longitude</code>	<code>float8</code>	NO	—	—	GPS longitude at capture (decimal degrees, WGS-84)
<code>altitude</code>	<code>float8</code>	YES	<code>NULL</code>	—	GPS altitude in metres (may be unavailable on some devices)
<code>gps_accuracy_metres</code>	<code>float8</code>	YES	<code>NULL</code>	—	Horizontal accuracy of GPS fix in metres

<code>captured_at</code>	<code>timestamp tz</code>	NO	—	—	Client-side capture timestamp (ISO 8601)
<code>accuracy_status</code>	<code>text</code>	NO	—	<code>CHECK IN ('precise', 'override')</code>	Whether GPS met the accuracy threshold or the driver overrode

Table 4.3.2 Table of ‘*delivery_records*’ relation stored in Supabase Backend

C. `field_test_records` Table — Column Dictionary

This table is specifically created for field testing, logging additional data such as GPS latency and battery level for post-test analysis, it is not used in production. Part of the table mirrors every production column for parity, Inserted by `sendToFieldTestAPI()` in `utils/field-test-api.ts`

Column Name	PostgreSQL Type	Nullable	Default	Description
— <i>all columns from <code>delivery_records</code> above</i> —				Production-parity fields
<code>model_confidence</code>	<code>float8</code>	NO	—	Building detector confidence score (0.0–1.0)
<code>building_detected</code>	<code>boolean</code>	NO	—	Whether the TFLite model classified a building above threshold
<code>detection_overridden</code>	<code>boolean</code>	NO	—	Whether the driver overrode the model's decision
<code>ground_truth_is_building</code>	<code>boolean</code>	YES	<code>NULL</code>	Manual annotation: was a building actually present? (for confusion matrix)
<code>barcode_scan_ms</code>	<code>int4</code>	YES	<code>NULL</code>	Time to complete barcode scan (milliseconds)
<code>gps_fix_ms</code>	<code>int4</code>	YES	<code>NULL</code>	Time to acquire GPS fix (milliseconds)
<code>time_to_capture_ms</code>	<code>int4</code>	YES	<code>NULL</code>	Total elapsed time from screen open to shutter press
<code>exif_write_ms</code>	<code>int4</code>	YES	<code>NULL</code>	Time to burn EXIF into image file

upload_latency_ms	int4	YES	NULL	Round-trip upload time (image + DB insert)
upload_success	boolean	NO	—	Whether the upload completed without error
battery_level	float8	YES	NULL	Device battery percentage at capture time
gps_speed	float8	YES	NULL	Device speed from GPS (m/s) for delivery path analysis
gps_heading	float8	YES	NULL	Device compass heading from GPS (degrees)

Table 4.3.3 Table of 'field_test_records' relation stored in Supabase Backend

D. Supabase Storage Buckets

The architecture incorporates two distinct Supabase Storage buckets: 'pod-images' for production delivery proof images and 'field-test-images' for images associated with model evaluation, thereby segregating data based on its operational context.

Bucket Name	Access	File Naming Convention	Content Type
delivery-proofs	Public read	delivery_{unix_ms}.jpg	image/jpeg
field-test-proofs	Public read	field_test_{unix_ms}.jpg	image/jpeg

Table 4.3.4 Table of Supabase Storage Buckets data design

4.3.2 Entity Relationship Diagram

Based on the codebase, the ER diagram should depict the following entities and relationships. Since the design don't have separate drivers or shipments tables yet (the current schema uses anonymous and implicit auth), the diagram reflects the actual implemented schema:

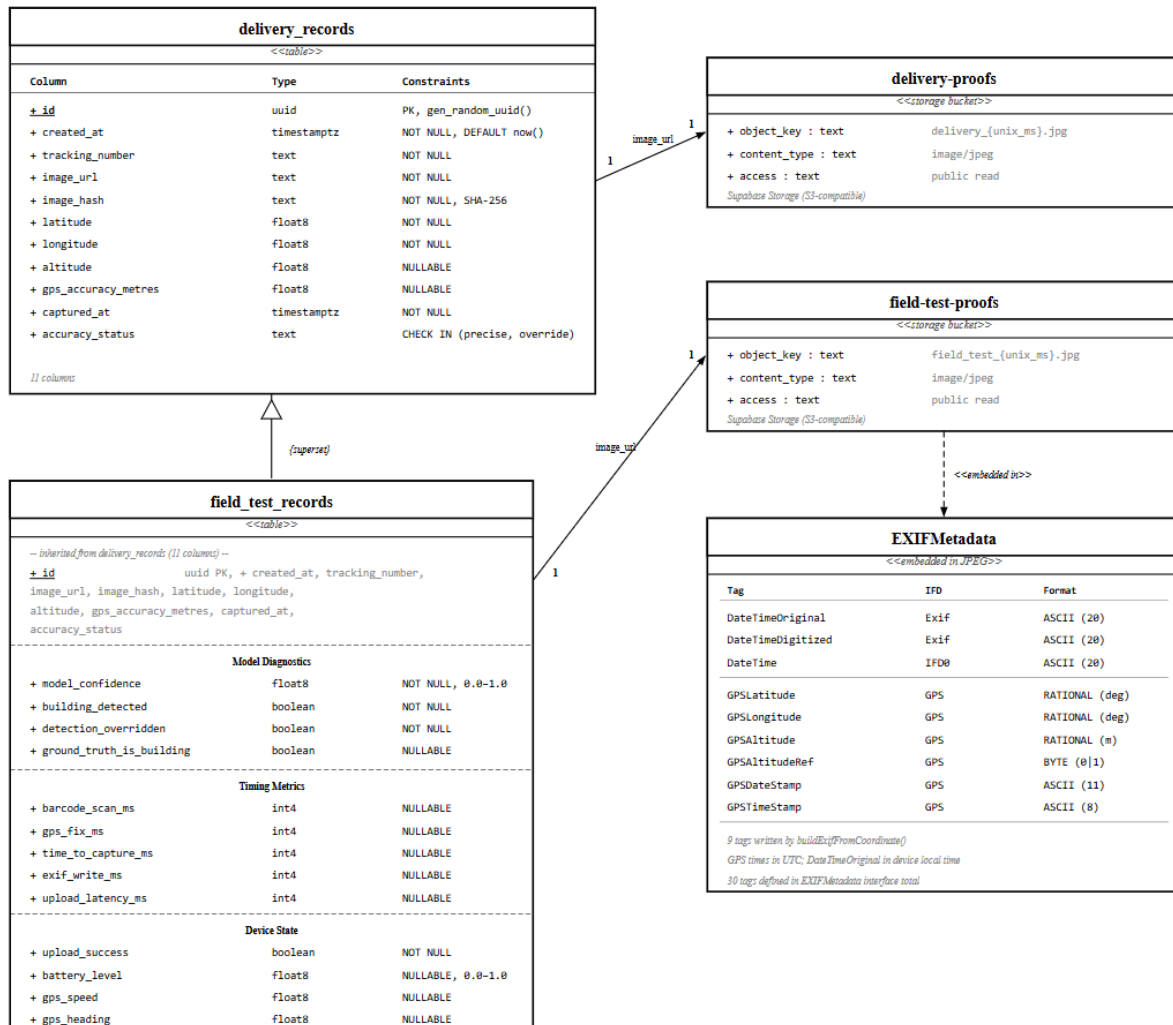


Figure 4.3.2 ERD Design of the Database

- **delivery_records.image_url** → references an object in the **delivery-proofs** storage bucket (1:1).
- **field_test_records.image_url** → references an object in the **field-test-proofs** storage bucket (1:1).
- **field_test_records** is structurally a **superset** of **delivery_records** — it contains all production columns plus diagnostic/evaluation columns. The two tables

are intentionally **denormalised** (no foreign key between them) to keep the field-test pathway isolated from production data.

- There is **no explicit drivers table** — the current system uses Supabase anonymous/implicit authentication. Driver identity is not stored per-record in the current implementation. If multi-driver support is added, a **drivers** table with FK relationships to both record tables would be introduced.

4.3.2 Related SQL Queries

This section shows the SQL queries to create tables for `delivery\_records`, `field\_test\_records`, explicitly defining column types, constraints, and default values as detailed in the preceding column dictionaries.

```
## Table `delivery_records`
CREATE TABLE public.delivery_records (
  id uuid NOT NULL DEFAULT gen_random_uuid(),
  created_at timestamp with time zone DEFAULT now(),
  image_url text NOT NULL,
  image_hash text NOT NULL,
  latitude double precision NOT NULL,
  longitude double precision NOT NULL,
  altitude double precision,
  gps_accuracy_metres double precision,
  captured_at timestamp with time zone NOT NULL,
  accuracy_status text NOT NULL CHECK (accuracy_status = ANY
  (ARRAY['precise'::text, 'override'::text])),
  tracking_number text,
  CONSTRAINT delivery_records_pkey PRIMARY KEY (id)
);
```

```
## Table `field_test_records`
CREATE TABLE public.field_test_records (
  id uuid NOT NULL DEFAULT gen_random_uuid(),
  image_url text,
  image_hash text,
  latitude double precision NOT NULL,
  longitude double precision NOT NULL,
  altitude double precision,
  gps_accuracy_metres double precision,
```

```
captured_at timestamp with time zone NOT NULL,  
accuracy_status text NOT NULL,  
tracking_number text NOT NULL,  
model_confidence double precision NOT NULL,  
building_detected boolean NOT NULL,  
detection_overridden boolean NOT NULL DEFAULT false,  
ground_truth_is_building boolean,  
barcode_scan_ms double precision,  
gps_fix_ms double precision,  
time_to_capture_ms double precision,  
upload_latency_ms double precision,  
exif_write_ms double precision,  
battery_level double precision,  
upload_success boolean DEFAULT false,  
gps_speed double precision,  
gps_heading double precision,  
created_at timestamp with time zone DEFAULT now(),  
CONSTRAINT field_test_records_pkey PRIMARY KEY (id)  
);
```

4.4 Front-End Interaction & Capture Modules

4.4.1 Dual-Layer Overlay & Barcode OCR

Functional Overview

The camera screen implements a dual-layer overlay that divides the live viewfinder into two spatial zones: a top half for building framing and a bottom half for barcode scanning. A single camera feed serves both purposes, with the system switching between a barcode code-scanner and a TFLite frame-processor depending on the current phase. The barcode scanner uses a stability-debounce algorithm to ensure only reliably decoded Code-128 values are accepted.

SENDER COPY

RECEIVER COPY

GDEX
GD EXPRESS SDN BHD
(347624-X)
www.gdexpress.com

CN25000907029

CBP

ACCOUNT NO: [REDACTED]
SENDER: GD Express Sdn Bhd
RECEIVER: [REDACTED]
ATTN: [REDACTED]
CONTACT: [REDACTED]

CBP CN
Print by: 67337
Print date: 10/03/2026

ORG: 46050
PJY
DEST: 46050

HQA

TYPE OF CONSIGNMENT:
☐ DOCUMENT
☒ PARCEL
☐ CHEQUE
NO. OF PC(S): 1
CONTENT: testimh
WEIGHT: 1
REMARK:

Figure 4.4.1 Example Consignment Note generated by GDEX

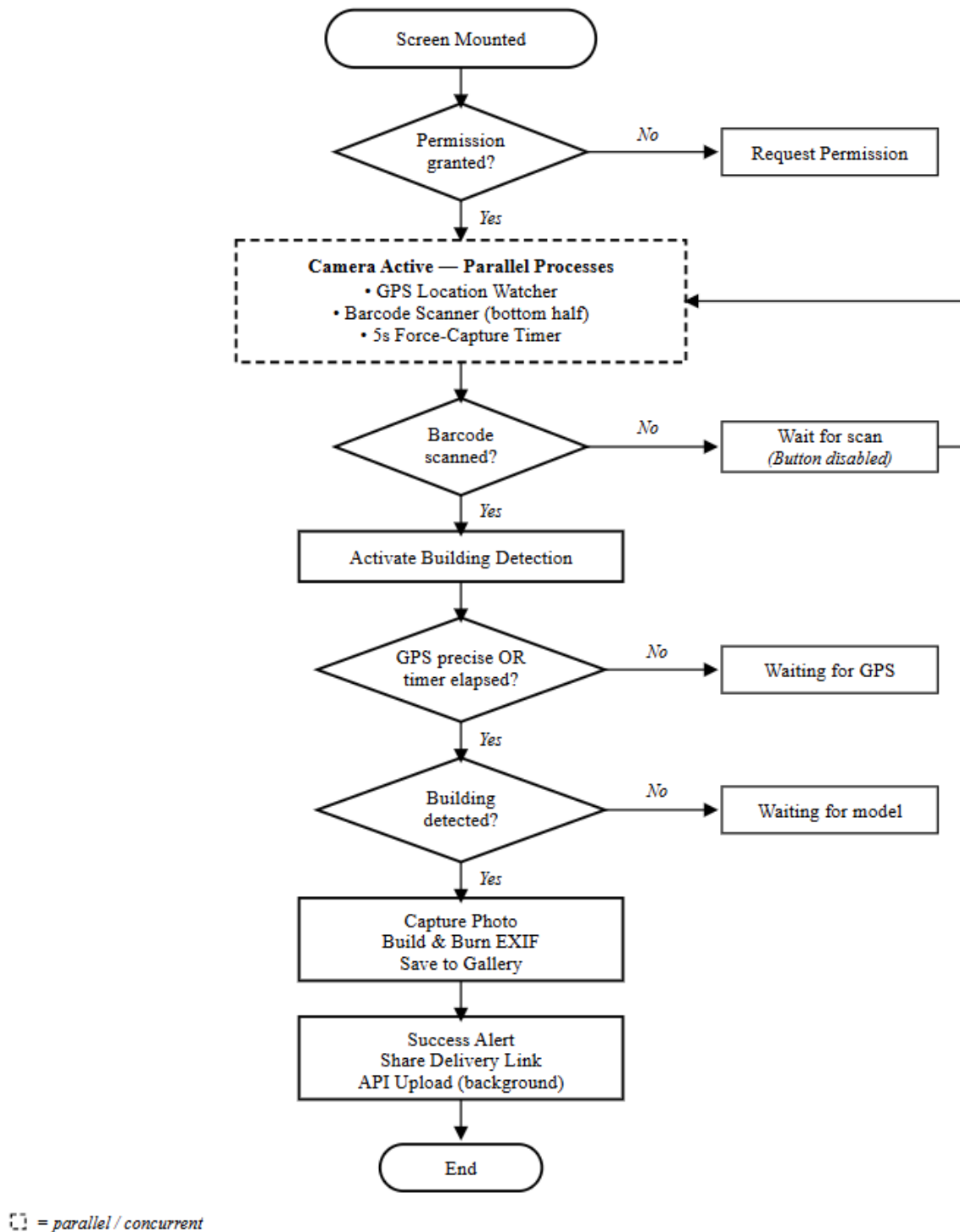


Figure 4.4.2 Flowchart of the Camera Capturing Procedure

Logic / Algorithmic Flow

A. Camera Mode Switching (Mutual Exclusion)

The `CameraView` component wraps `react-native-vision-camera`'s `Camera`. Only one processor can be active at a time. The camera screen conditionally passes either the `codeScanner` or the `frameProcessor` based on whether a tracking number has been locked:

```
// app/camera/index.tsx -- CameraView usage
<CameraView
  cameraRef={cameraRef}
  device={device}
  format={format}
  isActive={true}
  codeScanner={!trackingNumber ? codeScanner : undefined}
  frameProcessor={trackingNumber ? frameProcessor : undefined}
/>
```

- **Phase 1** — When `trackingNumber` is `null`, the `codeScanner` prop is active and the camera decodes barcodes on every frame.
- **Phase 2** — Once a barcode is locked, `codeScanner` becomes `undefined` and the TFLite `frameProcessor` activates for building detection.

This mutual exclusion prevents resource contention on the native camera pipeline.

B. Barcode Scanner Hook — `useBarcodeScanner()`

The scanning logic is encapsulated in a custom React hook located in `use-barcode-scanner.ts`. It returns a `codeScanner` object for the camera and exposes the scan state.

```
// features/camera/hooks/use-barcode-scanner.ts
const REQUIRED_STABLE_COUNT = 4;
const SUPPORTED_CODE_TYPES: CodeType[] = ['code-128'];

export function useBarcodeScanner(): UseBarcodeScannerReturn {
  const lastScannedDataRef = useRef<string | null>(null);
```

```

const stableCountRef    = useRef(0);
const isLockedRef       = useRef(false);
const [scanResult, setScanResult] = useState<BarcodeScanResult
| null>(null);
...
}

```

Key design decisions:

- **Code-128 only** — current `SUPPORTED_CODE_TYPES` is restricted to `['code-128']`, the standard symbology for logistics tracking numbers. All other barcode types are silently ignored.
- **Dual-track state management** — The hook maintains both React refs (for synchronous, closure-safe logic) and React state (for triggering UI re-renders). This prevents race conditions where stale closures could cause duplicate locks.

C. Stability Debounce Algorithm

The core scanning callback `handleCodeScanned` is invoked on every camera frame. Rather than accepting the first decoded value, it requires **4 consecutive identical readings** before locking:

```

// features/camera/hooks/use-barcode-scanner.ts -- handleCodeScanned
const handleCodeScanned = useCallback((codes: Code[]) => {
  if (isLockedRef.current) return; // Already locked -- short-circuit

  for (const code of codes) {
    if (code.type !== 'code-128' || !code.value) continue;
    if (code.value === lastScannedDataRef.current) {
      stableCountRef.current += 1;
      if (stableCountRef.current >= REQUIRED_STABLE_COUNT) {
        isLockedRef.current = true; // Ref: prevent further callbacks
        setScanResult({ // State: trigger UI re-render
          value: code.value,
          type: code.type,
          corners,
        });
        return;
      }
    } else {
      lastScannedDataRef.current = code.value;
      stableCountRef.current = 1; // New candidate -- reset counter
    }
  }
}

```

```
    }  
  }, []);
```

The stability threshold of 4 frames prevents false positives caused by partially decoded barcodes, motion blur, or flickering values when the parcel is at an oblique angle.

D. Scan Reset

The hook exposes a `resetScan()` method that clears all refs and state, returning the camera to **Phase 1** defined in section A:

```
// features/camera/hooks/use-barcode-scanner.ts -- resetScan  
const resetScan = useCallback(() => {  
  lastScannedDataRef.current = null;  
  stableCountRef.current = 0;  
  isLockedRef.current = false;  
  setScanResult(null);  
}, []);
```

This is triggered by the "Rescan" button in the overlay UI, allowing the user to correct a misread without leaving the camera screen.

E. BarcodeViewfinder — Visual Feedback Component

The `BarcodeViewfinder` component renders an animated viewfinder rectangle using `react-native-reanimated`. It accepts an `isDetected` boolean and transitions between two visual states:

```
// features/camera/components/BarcodeViewfinder.tsx  
interface BarcodeViewfinderProps {  
  isDetected: boolean;  
  width?: number;  
  height?: number;  
  style?: ViewStyle;  
}
```

- **Idle state** (`isDetected = false`): Four corner brackets in distinct colours pulse in opacity via `withRepeat`. A vertical scan line sweeps horizontally (1.4 s per direction) to indicate active scanning.

- **Detected state** (`isDetected = true`): Corners turn green (`#22c55e`), the viewfinder box shrinks to 82% scale via a spring-like `withSequence`, and all repeating animations stop. This lock-on animation provides immediate visual confirmation.

```
// BarcodeViewfinder.tsx -- Lock-on animation
boxScale.value = withSequence(
  withTiming(0.75, { duration: 200, easing:
    Easing.out(Easing.ease) }),
  withTiming(0.82, { duration: 100 })
);
pulse.value = withTiming(1, { duration: 150 });
```

F. Overlay Layout — Dual-Zone Structure

The overlay is an absolutely positioned `View` layered on top of the camera. It is divided by a horizontal separator:

```
// app/camera/index.tsx -- Overlay structure
<View className="absolute inset-0" pointerEvents="box-none">
  /* Top Half -- Building zone */
  <View style={{ flex: 1 }}>
    /* GPS accuracy badge, building confidence, "CAPTURE
    BUILDING HERE" label */
  </View>

  /* Middle Separator */
  <View style={styles.separatorLine} />

  /* Bottom Half -- Parcel zone */
  <View style={{ flex: 1 }}>
    <BarcodeViewfinder isDetected={!!trackingNumber} />
    /* Tracking number display or "Point at Code 128 barcode"
    instruction */
    /* Capture button */
  </View>
</View>
```

Both halves use `flex: 1` to equally split the screen. Corner brackets (`border-t-2 border-l-2`, etc.) frame each zone to guide user alignment.

Data Interaction

Direction	Data	Description
Input	Live camera frames	Streamed to <code>useCodeScanner</code> via <code>react-native-vision-camera</code>
Input	GPS coordinates	From <code>useLocationWatcher()</code> : accuracy determines badge colour
Output	BarcodeScanResult	<code>{ value, type, corners }</code> : the locked barcode string, symbology type, and corner coordinates
Output	Camera mode switch	<code>codeScanner</code> disabled, <code>frameProcessor</code> activated
Output	UI state	Viewfinder animation, instructional labels, capture button enabled/disabled

Table 4.4.1 Data Integration design of the Camera View

Design Constraints & Exceptions

- **Code-128 only** — Other type of machine-readable optical labels are silently skipped; the scanner does not produce false matches from QR codes or other formats on the parcel.
- **No manual entry fallback** — If the barcode is unreadable (damaged label, extreme angle), the user must physically reposition the parcel. No manual tracking-number input exists in the current design.
- **Capture gating** — The capture button is disabled until all prerequisites are met: barcode locked, GPS fix (green ≤ 10 m or force-override after 5 s), and building detected (or test-mode override).
- **Force-capture timer** — If GPS accuracy does not reach green within 5 seconds, a fallback enables "Force Capture (Low Accuracy)", tagged as `override` in the payload for downstream flagging.

4.4.2 POD Landing Page:

The Proof-of-Delivery (POD) landing page is a client-facing web application that allows parcel recipients to view and verify delivery records associated with their parcels. It is deployed as a publicly accessible Next.js page at the route `/delivery`, and is accessed via a URL containing a query parameter `id` that identifies a specific delivery record (e.g., `/delivery?id=<uuid>`). A special value of `id=latest` retrieves the most recently created record, serving as a convenient shortcut during testing and demonstrations. The Landing Page is hosted on Vercel, which connects to a GitHub repository for rendering.

The page provides four core functions:

- **Delivery record retrieval** — On page load, the application fetches a single delivery record from the Supabase `delivery_records` table, presenting key metadata including the tracking number, capture timestamp, GPS coordinates, GPS accuracy, and accuracy status.
- **Proof-of-delivery photo display** — The delivery photo captured by the driver is rendered directly on the page, sourced from a public image URL stored in the database.
- **Image download with EXIF metadata** — A download button allows the recipient to save the original proof-of-delivery image locally, preserving any embedded EXIF metadata (e.g., camera and geolocation data) for independent offline verification.
- **Image integrity verification** — The page implements client-side SHA-256 hash verification to detect image tampering. The verification function fetches the image, converts it to a Base64 string representation (matching the hashing method used by the mobile capture application), computes its SHA-256 digest using the Web Crypto API, and compares it against the hash stored in the database at upload time. The result is displayed to the recipient as either **"Image is authentic and unmodified"** or **"Image has been tampered with"**.

Table 4.4.2 below shows the Visual Design of the POD Landing Page

Section	Design Token
Page background	White canvas — no dark mode
Hero header	display-lg (40px / weight 400), ink text
Tracking number	cream-callout-card — signature-cream bg, rounded-md , 24px padding
Photo container	rounded-lg with hairline border
Download button	button-primary — near-black bg, white text, rounded-lg
Details card	feature-card-tabbed — surface-soft bg, rounded-lg , 32px padding, hairline dividers
Integrity section	hero-card-dark — surface-dark bg, rounded-lg , 48px padding
Verify button	button-secondary-on-dark — white bg on dark card
Links	link color (#1b61c9)
Loading/error/fallback	Canvas background with muted text

```
// Delivery Record Fetching
const fetchData = async () => {
  console.log("Fetching for id:", id);

  let query = supabase.from("delivery_records").select("tracking_number");

  // If id is 'latest' or not provided, fetch the most recent record
  // Otherwise, fetch the record with the specified id
  if (id === "latest" || !id) {
    query = query.order("created_at", { ascending: false }).limit(1);
  } else {
    query = query.eq("id", id);
  }

  const { data, error } = await query
    .select(
      "id, tracking_number, image_url, image_hash, latitude, longitude,
      gps_accuracy_metres, captured_at, accuracy_status",
    )
    .single();

  console.log("data:", data);
}
```

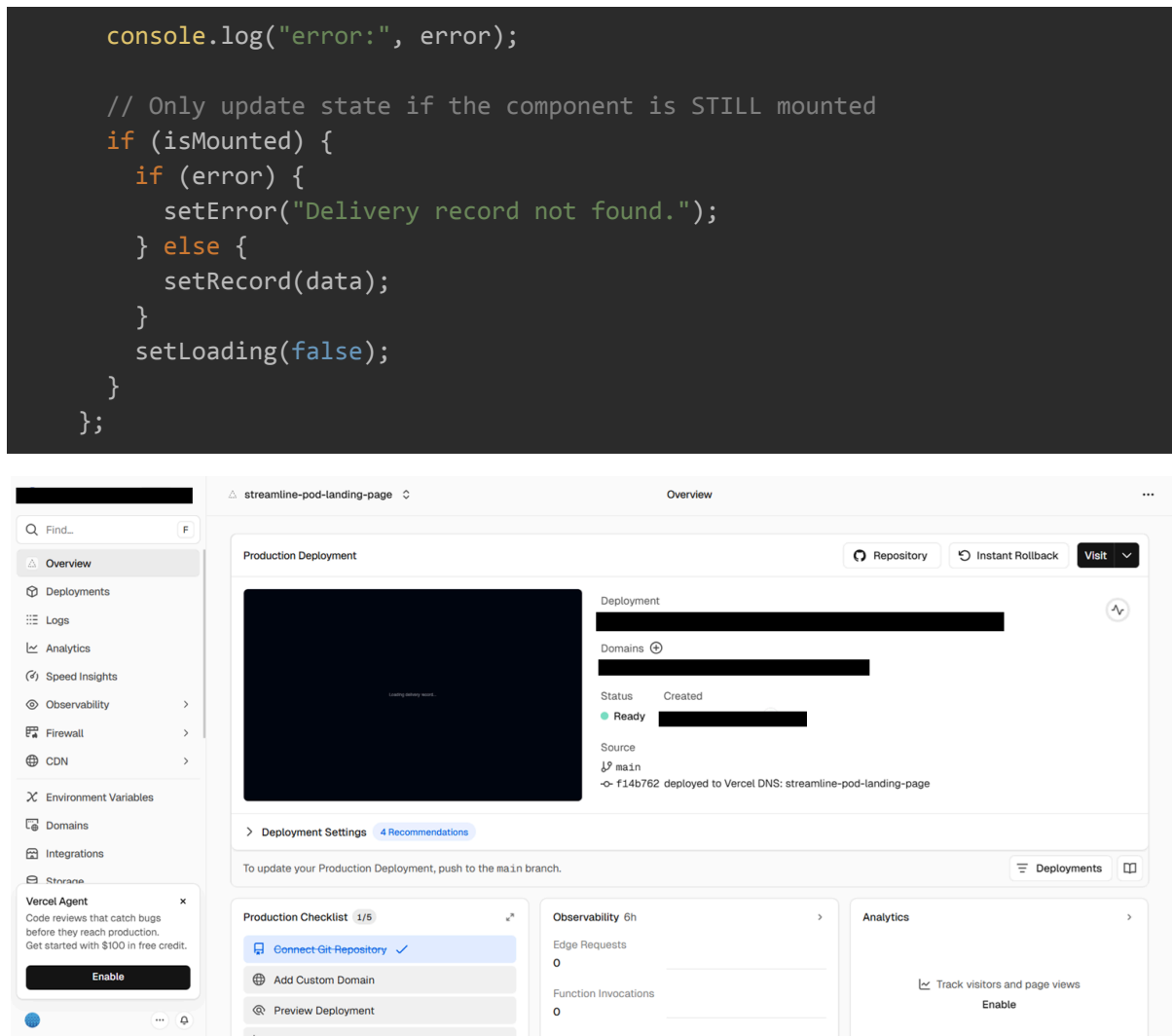


Figure 4.4.3 shows the Dashboard on Vercel for deploying the Landing Page

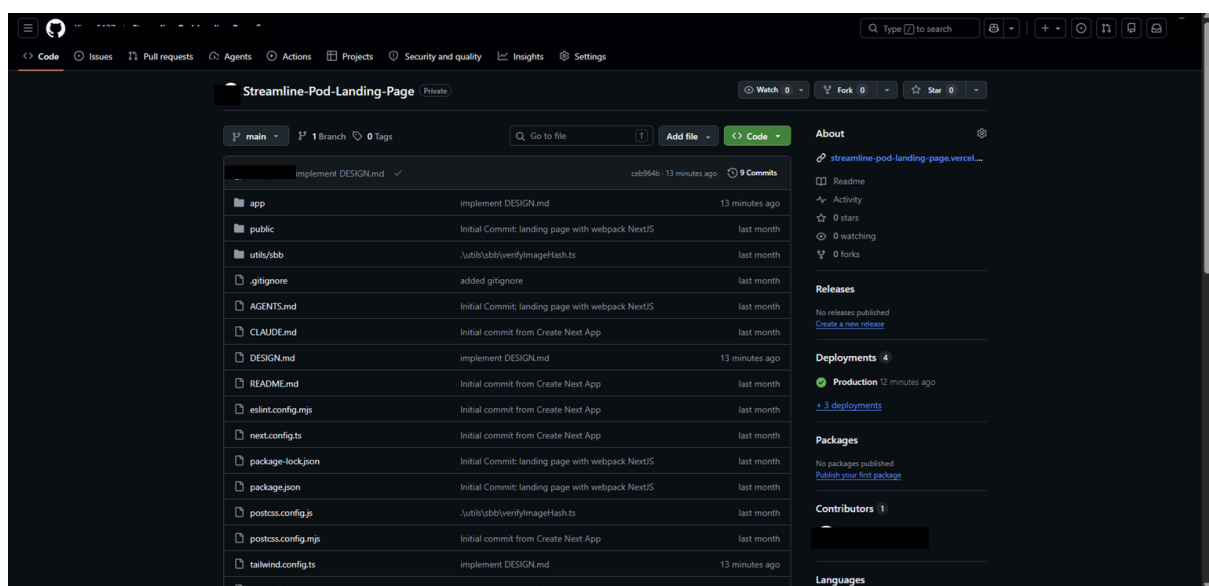


Figure 4.4.4 shows the GitHub Repository of the Landing Page

4.5 Verification & Metadata Logic

4.5.1 Geolocation Gating Algorithm

Functional Overview

The geolocation gate ensures that proof-of-delivery photos are only captured when the GPS fix is precise enough to be trusted. It implements a **traffic-light accuracy signal** (green / yellow / red) coupled with a **5-second fallback timer** that grants an explicit override when the signal fails to reach green within a reasonable wait. The captured coordinate is later burned into the photo's EXIF metadata for tamper-evident verification.

Logic / Algorithmic Flow

A. Accuracy Traffic-Light Signal — useAccuracySignal()

The signal is derived purely from the device's reported **accuracy** value in metres returned by **expo-location**'s **watchPositionAsync** with **BestForNavigation**:

```
// features/camera/hooks/use-accuracy-signal.ts
const signal: AccuracySignal =
  accuracy !== null && accuracy <= 10
    ? 'green'
    : accuracy !== null && accuracy <= 50
      ? 'yellow'
      : 'red';

const isGreen = signal === 'green';
```

The thresholds (**10 m** and **50 m**) reflect a deliberate compromise between operational realism and evidentiary value:

```
Rule expression: canCapture = !!location && !!trackingNumber &&
(isGreen || forceCaptureEnabled) && (isTestMode ||
isBuildingDetected)
```

	GPS Signal	Accuracy	GPS Fix Exists	Force Timer (5 s)	Building AI ($\geq 80\%$)	canCapture	Button Label
1	 Green	≤ 10 m	Yes	Cleared	 Yes	 Allowed	Verified Capture
2	 Green	≤ 10 m	Yes	Cleared	 No	 Denied	Verified Capture (disabled)
3	 Yellow	10–50 m	Yes	Not expired	 Yes	 Denied	Improving Accuracy...
4	 Yellow	10–50 m	Yes	Not expired	 No	 Denied	Improving Accuracy...
5	 Yellow	10–50 m	Yes	Expired	 Yes	 Allowed	Force Capture (Low Accuracy)
6	 Yellow	10–50 m	Yes	Expired	 No	 Denied	Force Capture (Low Accuracy) (disabled)
7	 Red	> 50 m	Yes	Not expired	 Yes	 Denied	Improving Accuracy...
8	 Red	> 50 m	Yes	Not expired	 No	 Denied	Improving Accuracy...
9	 Red	> 50 m	Yes	Expired	 Yes	 Allowed	Force Capture (Low Accuracy)
10	 Red	> 50 m	Yes	Expired	 No	 Denied	Force Capture (Low Accuracy) (disabled)
11	 Red	No fix	No	Not expired	Any	 Denied	Improving Accuracy...
12	 Red	No fix	No	Expired	Any	 Denied	Force Capture (Low Accuracy) (disabled)

Table 4.5.1 Accuracy Threshold Decision Table

B. Force-Capture Fallback Timer

In dense urban environments (terraced housing, covered porches, multi-storey blocks), the device may never report ≤ 10 m accuracy. To prevent the driver from being blocked indefinitely, a 5-second timer in the Capture Screen enables an explicit override:

```
// app/camera/index.tsx -- Force-capture timer
useEffect(() => {
  if (isGreen) {
    clearTimeout();
    setForceCaptureEnabled(false);
    return;
  }
  if (!isGreen && !timerRef.current) {
    timerRef.current = setTimeout(() => {
      setForceCaptureEnabled(true);
    }, 5000);
  }
  return clearTimeout;
}, [isGreen]);
```

The capture button's enable state is then a conjunction of the gate plus the other prerequisites:

```
const canCapture =
  !!location &&
  !!trackingNumber &&
  (isGreen || forceCaptureEnabled) &&
  (isTestMode || isBuildingDetected);
```

When force-capture is used, the payload is tagged `accuracyStatus: 'override'` instead of `'precise'`, allowing the back-office system to flag the delivery for review.

Direction	Data	Description
Input	<code>accuracy</code> (m)	Reported by <code>expo-location</code> on every position update
Input	<code>isGreen</code> flag	Derived from accuracy threshold
Output	<code>signal</code> , <code>badgeColor</code> , <code>badgeText</code> , <code>buttonLabel</code>	UI feedback consumed by the overlay

Output	<code>forceCaptureEnabled</code>	Boolean unlocking the capture button after 5 s
Output	<code>accuracyStatus</code> field	' <code>precise</code> ' or ' <code>override</code> ' written into the API payload

Table 4.5.2 Data Integration design of the geolocation algorithm

Design Constraints & Exceptions

- **No location permission** — `useLocationWatcher` surfaces a permission error; the GPS badge displays the error message and `canCapture` remains `false`.
- **Stale fix** — The watcher uses a 1-second `timeInterval` and 1-metre `distanceInterval`, so the latest coordinate is rarely more than a second old.
- **Override abuse** — Because every override is tagged in the payload, drivers cannot silently bypass the gate; persistent overrides for a given route surface in operational reports.

4.5.2 EXIF Metadata Injection Logic:

Functional Overview

The Metadata Injector stamps each captured photo with a **tamper-evident spatial and temporal fingerprint** before the image is persisted to the device gallery. It transforms a live `LocationCoordinate` snapshot into standards-compliant EXIF tags and writes them directly into the JPEG's APP1 segment, producing a self-contained evidentiary artefact that survives sharing, re-encoding, and archival.

Logic / Algorithmic Flow

The injector is a two-stage pipeline split across two pure modules:

A. Stage 1 — Build the EXIF Object (`buildExifFromCoordinate`)

Located in `exif-builder.ts`, this pure function maps a `LocationCoordinate` into an `EXIFMetadataRecord`. The mapping honours the EXIF 4.3.1 specification's split between **local** capture time and **UTC** GPS time:

```
// utils/exif-builder.ts
export function buildExifFromCoordinate(coordinate: LocationCoordinate):
EXIFMetadata {
    const capturedAt = new Date(coordinate.timestamp);
    const exifDateTime = toExifDateTime(capturedAt);

    return {
        GPSLatitude:      coordinate.latitude,
        GPSLongitude:     coordinate.longitude,
        GPSAltitude:      coordinate.altitude ?? undefined,
        GPSAltitudeRef:   coordinate.altitude !== null && coordinate.altitude <
0 ? 1 : 0,
        GPSTimeStamp:     toGpsTimeStamp(capturedAt),    // UTC -- "YYYY:MM:DD"
        GPSTimeStamp:     toGpsTimeStamp(capturedAt),    // UTC -- "HH:MM:SS"
        DateTimeOriginal: exifDateTime,                  // local -- capture
instant
        DateTimeDigitized: exifDateTime,                  // local -- file
creation
        DateTime:         exifDateTime,                  // local -- last
modified
    };
}
```

Field mapping at a glance:

Source field	EXIF tag	Notes
<code>coordinate.latitude</code>	GPSLatitude	Signed decimal degrees
<code>coordinate.longitude</code>	GPSLongitude	Signed decimal degrees
<code>coordinate.altitude</code>	GPSAltitude GPSAltitudeRef	+ Ref=1 if below sea level
<code>coordinate.timestamp</code>	GPSTimeStamp GPSTimeStamp	/ Formatted as UTC per spec
<code>coordinate.timestamp</code>	DateTimeOriginal ...Digitized / DateTime	/ Formatted as device local time

Table 4.5.3 Table show field mapping of the value sources to EXIF tags

B. Stage 2 — Burn the Tags into the File (burnExifOnly)

Located in `exif-processor.ts`, this wraps `@lodev09/react-native-exify`'s native `writeAsync`, which performs an in-place APP1-segment merge on the JPEG:

```
// utils/exif-processor.ts
export async function burnExifOnly(
  photoUri: string,
  exifTags: EXIFMetadata
): Promise<{ uri: string }> {
  const writeResult = await writeAsync(photoUri, exifTags as any);
  return { uri: writeResult?.uri ?? photoUri };
}
```

A sibling helper, `saveToGalleryOnly()` is invoked *after* burning so the gallery receives the already-stamped file. This ordering matters on iOS, where EXIF cannot be written into an Asset URI — the injector must run on the raw `file://` output of the camera *before* `MediaLibrary.createAssetAsync` takes over.

C. Call-Site Orchestration

The camera screen calls the two stages in sequence immediately after capture:

```
// app/camera/index.tsx -- handleCapture
const photo          = await capturePhoto();
const exifMetadata   = buildExifFromCoordinate(location); // Stage 1
const written        = await burnExifOnly(photo.uri, exifMetadata); // Stage 2
await saveToGalleryOnly(written.uri, 'Streamline');
```

The write step is timed (`exifWriteMs`) and surfaced in the Field Test payload for later performance analysis.

Direction	Data	Description
Input	<code>photo.uri</code>	Local <code>file://</code> URI of the raw JPEG from the camera
Input	<code>LocationCoordinate</code>	Latest GPS snapshot (lat, lon, altitude, timestamp)

Output	Burned JPEG at <code>written.uri</code>	Same file with EXIF APP1 segment merged
Output	<code>exifWriteMs</code>	Timing metric used by field-test telemetry

Table 4.5.4 shows the Data Interaction design of the EXIF injection algorithm flow

Design Constraints and Exceptions

- **Pure builder, impure writer** — `buildExifFromCoordinate()` is a pure function with no I/O, making it trivially unit-testable. All native/filesystem concerns are isolated in `burnExifOnly()`.
- **Platform quirk** — On iOS, the write must target the raw camera output; on Android it can be in-place. Keeping the gallery save in a separate function enforces the correct order on both platforms.
- **Graceful degradation** — If `writeAsync` throws, `handleCapture` logs the failure and continues with the un-stamped URI rather than aborting the whole capture (see the `try/catch` in `index`). The operation is best-effort; losing an EXIF stamp is preferable to losing the whole photo.
- **OCR field not injected (current scope)** — Tracking number is sent via the API payload only; extending the schema to an `ImageDescription` EXIF tag would be a future enhancement.

4.6 Machine Learning Model: Building Detection

This section delineates the complete developmental lifecycle of the edge-native semantic validation model. It details the methodologies employed to transition from manual, ambiguous visual capture guidelines to an autonomous, mathematically verifiable deep learning pipeline. The ensuing sub-sections will explicitly break down the empirical data curation strategy, the algorithmic preprocessing pipeline, model training via Neural Architecture Search (NAS), and the final asynchronous integration into the mobile architecture. The Platform used to train the model is via Google Vertex AI Platform.

4.6.1 Empirical Data Curation and Preprocessing

The primary objective of this module is to algorithmically validate that a captured Proof of Delivery (POD) photograph contains legitimate architectural structures, thereby mitigating fraudulent or inaccurate delivery logging. Historically, logistics personnel have relied on legacy, static POD capture guidelines provided by GDEX, which often resulted in high variance and human error.

To systematically train a robust classification model, the environmental scope required strict bounding. Following consultations with GDEX stakeholders, the initial training domain was scoped specifically to Malaysian "terrace housings" and "shoplots." Because no standardized, open-source dataset exists that accurately categorizes this specific regional architecture, a proprietary dataset was manually curated via in-situ field photography across the Kampar, Perak district.

Real-World Environmental Variance and Object Injection

In operational contexts, deliveries are not invariably positioned at the outermost gate; parcels are commonly placed within interior foyers or porches. Accordingly, the dataset was constructed to encompass both macro-scale exterior architectural features and micro-scale foyer environments.

To prevent the neural network from classifying delivery parcels as buildings, an "object injection" data augmentation strategy was used. Standardized white and brown parcels were added to images against varied architectural backdrops, the network was constrained to treat them as extraneous, variable artifacts, thereby confining its learned parameters to immutable

architectural elements . Dataset robustness was enhanced with diverse lighting, occlusions, and viewpoints matching real field conditions.



Figure 4.6.1 Sample White and Brown parcel to be introduced in training image datasets

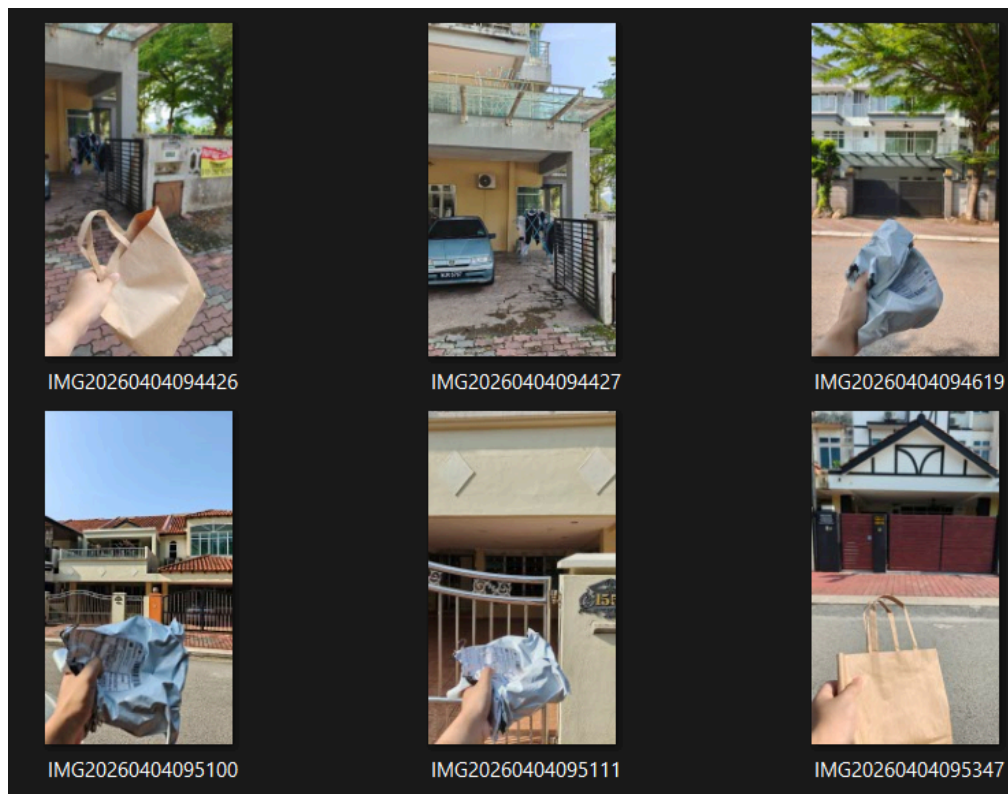


Figure 4.6.2 shows 6 samples of the curated training datasets, with alternating flows between no parcels, brown parcels, and white parcels

Guideline on Photo POD



Figure 4.6.3 shows the Guidelines from GDEX on taking photo POD images, the angle and objects shown is used as references for the dataset collection process

Categorization and Hard Negative Mining

To optimize the classification boundaries, the dataset was meticulously stratified into four distinct classes, the example of which is shown in Figure 4.6.4

- **terrace_housing** - shown in **RED**
- **terrace_foyer** - shown in **GREEN**
- **negatives_clean** - shown in **YELLOW**
- **negatives_hard** - shown in **BLUE**

The inclusion of deliberate negative datasets is a critical requirement for a robust binary classifier:

- **Clean Negatives:** This subset comprises purely environmental backgrounds (e.g., open skies, empty asphalt roads, dense foliage). These establish a baseline, teaching the model to reject non-man-made environments.
- **Hard Negatives:** This subset is crucial for mitigating false positives in production. It consists of isolated man-made structures, such as standalone chain-link fences,

unattached brick walls, or generic metal textures that share low-level visual features with houses but lack the complete semantic structure of a valid delivery location.



Figure 4.6.4 shows 3 sample datasets collected for each categories

Python Script in Pre-processing to prepares a custom image classification dataset

Prior to cloud ingestion and fine-tuning via Google Vertex AI (AutoML Vision), the raw photographic data required strict dimensional and metadata standardization because it consisted of varying resolutions ranging from 3 to 12 Megapixels, inconsistent EXIF orientation flags (landscape versus portrait discrepancies), mixed color modes, and non-standardized OS-generated filenames (e.g., `IMG20260420112752.jpg`).

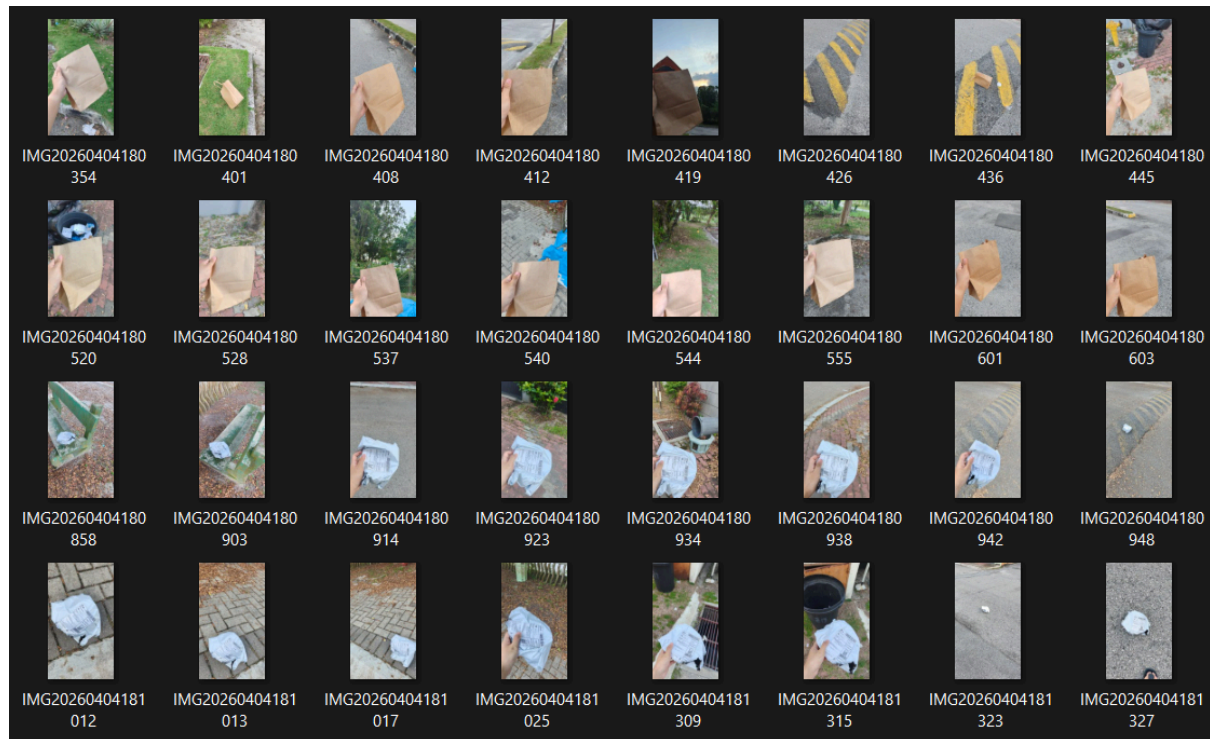


Figure 4.6.5 shows example of unstandardized raw datasets uploaded to the training device

For every raw image in every mapped source folder, the script performed five operations:

1. **EXIF Orientation Fix** — `ImageOps.exif_transpose(img)` physically rotates pixels so portrait/landscape orientation is baked in (prevents models from seeing sideways images).
2. **Color Standardization** — `img.convert('RGB')` strips alpha/transparency and forces 3-channel RGB.
3. **Aspect-preserving Resize** — `img.thumbnail((1024, 768), LANCZOS)` shrinks each image to fit inside a 1024×768 box without distortion, using high-quality LANCZOS resampling.
4. **Rename to a clean sequential scheme** — e.g.

- terrace_housing_001.jpg,
- negatives_hard_047.jpg.

A global counter per output class guarantees unique, contiguous numbering even when multiple source folders are merged into one class. The core code blocks are shown below.

Codeblock Configuration and class merging map:

```
# --- Configuration ---
SOURCE_DIR = 'raw_data'
TARGET_DIR = 'processed_dataset'
TARGET_SIZE = (1024, 768) # Optimized for Vertex AI speed/quality
QUALITY = 85 # Balance between file size and detail

# Maps each source folder → output folder name.
# Multiple source folders mapping to the same output folder are merged.
CATEGORY_MAP = {
    'terrace_housings': 'terrace_housing',
    'terrace_foyers': 'terrace_foyer',
    'negatives_hard': 'negatives_hard',
    'negatives_clean': 'negatives_clean',
    'neg_shoplot': 'negatives_hard', # merged into negatives_hard
    'pos_shoplot': 'pos_shoplot',
}

The image transformation pipeline:
try:
    with Image.open(img_path) as img:
        # 1. FIX ORIENTATION: Uses EXIF data to physically rotate pixels
        img = ImageOps.exif_transpose(img)

        # 2. STANDARDIZE COLOR: Removes transparency/alpha channels
        img = img.convert('RGB')

        # 3. RESIZE: Maintains aspect ratio within the 1024x768 box
        img.thumbnail(TARGET_SIZE, Image.Resampling.LANCZOS)

        # 4. RENAME: Professional naming (e.g., negatives_hard_001.jpg)
        new_name = f"{out_folder}_{global_index:03d}.jpg"
        save_path = os.path.join(target_category_path, new_name)

        # 5. SAVE: Compressed JPEG
        img.save(save_path, "JPEG", quality=QUALITY)
        global_index += 1
```

The empirical curation yielded a highly targeted dataset consisting of **435 positive architectural instances** (predominantly 319 exterior configurations and 114 foyer

environments) and **143 verified negative instances** (comprising 69 clean environmental backgrounds and 74 structurally ambiguous hard negatives).

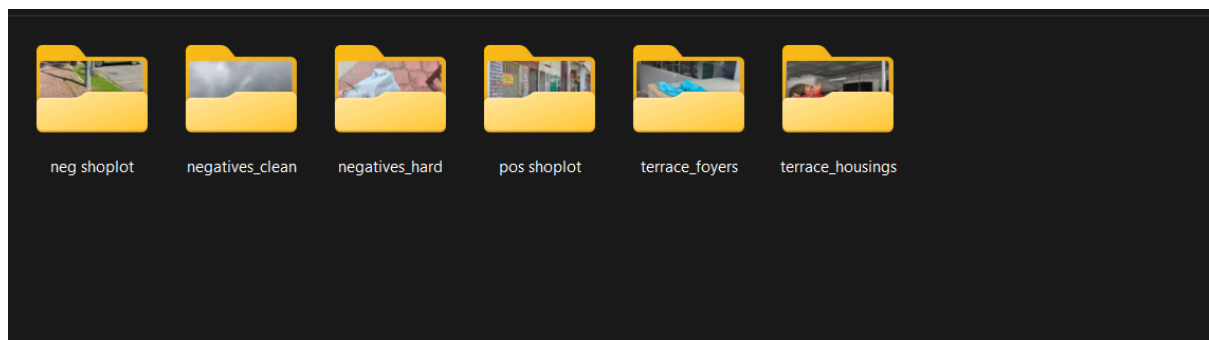


Figure 4.6.6 shows the different sub-folders of categorization of training datasets after pre-processing

4.6.2 Transfer Learning via Neural Architecture Search (NAS)

Following the algorithmic standardization of the dataset, the empirical data was ingested into the Google Vertex AI pipeline to establish ground-truth labels and execute the fine-tuning protocol.

Semantic Annotation and Ground Truth Establishment

The standardized spatial tensors were uploaded to the Google Vertex AI Cloud Console for manual semantic annotation. Each image was explicitly mapped to one of two labels:

- **building_detected** (comprising **terrace_housing** and **terrace_foyer**)
- **no_building** (comprising **negatives_clean** and **negatives_hard**)

This strict ground-truth labeling is the fundamental prerequisite for supervised classification, allowing the subsequent training algorithm to calculate accurate gradient updates.

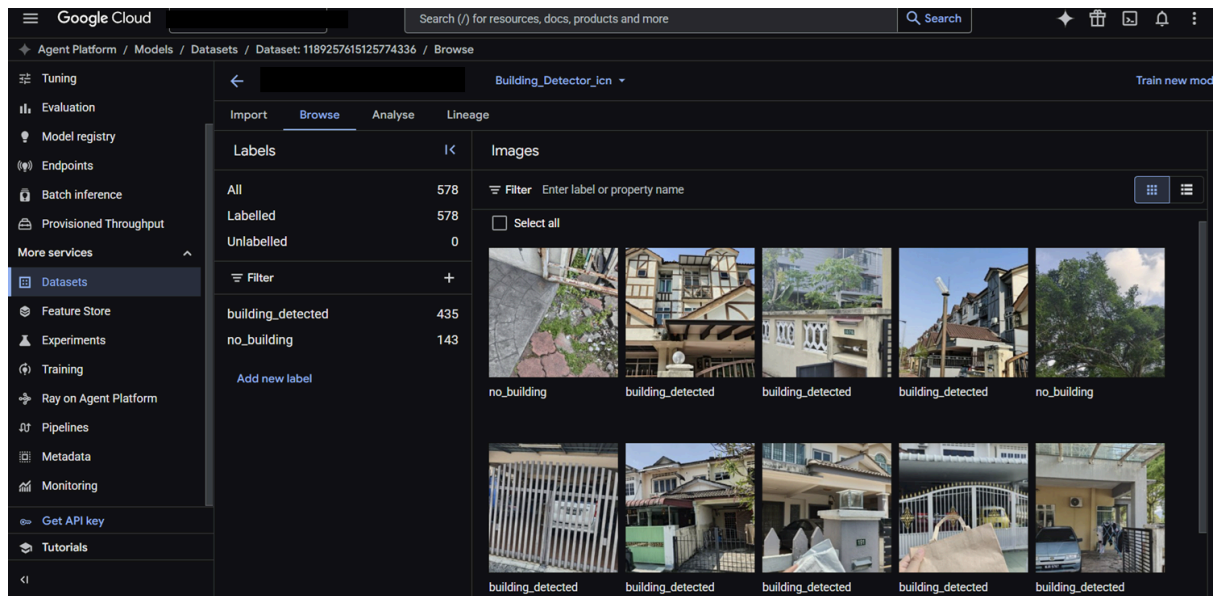


Figure 4.6.2 shows the training dashboard from Vertex Google AI Platform

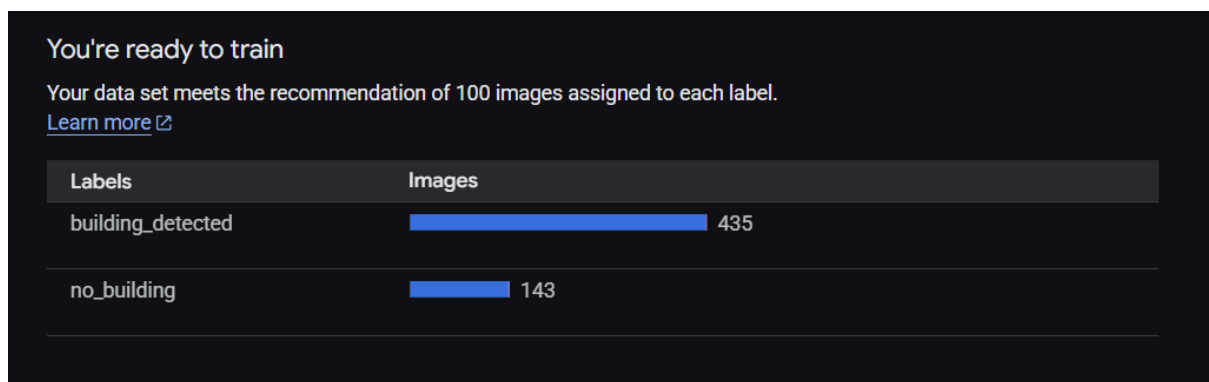


Figure 4.6.3 shows the training datasets quantity

As illustrated in the figure above, the Building Recognition model was trained on 578 labelled datasets, of which:

- 435 **building_detected** labels, consisting
 - 319 images of terrace housing exteriors
 - 116 images of terrace foyers
- 143 **no_building** labels, consisting
 - 69 clean negatives
 - 74 hard negatives

Architectural Justification

Neural Architecture Search (NAS) vs. Static Fine-Tuning

Traditional transfer learning methodologies typically require a machine learning engineer to manually instantiate a static, pre-trained base graph (e.g., a standard PyTorch implementation of MobileNetV3 or ResNet-50) and blindly fine-tune the dense layers. This manual approach frequently results in sub-optimal parameter utilization and excessive memory footprints when deployed to mobile Edge devices.

To achieve superior operational efficiency, this system abandons manual static architecture selection in favor of Google Vertex AI's AutoML Vision pipeline. The primary architectural advantage of AutoML is its deployment of **Neural Architecture Search (NAS)**.

Rather than statically fine-tuning a predefined model, NAS autonomously analyzes the proprietary Kampar dataset. It explores a multi-dimensional search space to identify the most efficient neural topology, thus optimizing convolutional layer depths, activation functions, and pruning for mobile deployment, all before fine-tuning. This guarantees a model architecture that is mathematically bespoke to the dataset.

Figure 4.6.4, 4.6.5 shows the results report and metrics of the trained model under NAS methodology

Classification Optimization Metric

Once the optimal topology is defined by the NAS algorithm, the supervised training loop commences. The network adjusts its convolutional weights iteratively by mathematically minimizing the discrepancy between its probabilistic predictions and the established ground-truth labels. For this multi-class classification task, the optimization is governed by the standard Categorical Cross-Entropy Loss function:

$$L = - \sum_{i=1}^C y_i \log(\hat{y}_i)$$

Where:

- C represents the total number of distinct classification categories ($C = 4$).
- y_i is the binary indicator (0 or 1) representing whether class is the true, annotated label for the current input tensor.
- $\hat{y}_i$ is the model's predicted probability (between 0.0 and 1.0) that the input tensor belongs to class i .

By continuously minimizing this loss vector over multiple epochs, the edge-native model learns to confidently isolate critical architectural features (e.g., gates, structural pillars) while mathematically ignoring the injected operational noise (such as the delivery parcels).

4.6.3 Edge Quantization for Mobile Development

Deploying raw, unoptimized neural networks to mobile devices is architecturally constrained by hardware memory and latency limits. To enable sub-10ms asynchronous inference on the driver's local hardware, the model underwent Post-Training Quantization prior to its export as a TensorFlow Lite (`.tflite`) payload.

Quantization mathematically compresses the network by reducing the precision of its learned weights and activation functions. Specifically, the model's parameters were downscaled from

32-bit floating-point values (**float32**) to 8-bit integers (**int8**). This precision reduction drastically shrinks the static file size and minimizes the runtime memory footprint, allowing the model to be executed natively by the mobile device's Neural Processing Unit (NPU) without suffering catastrophic accuracy degradation.

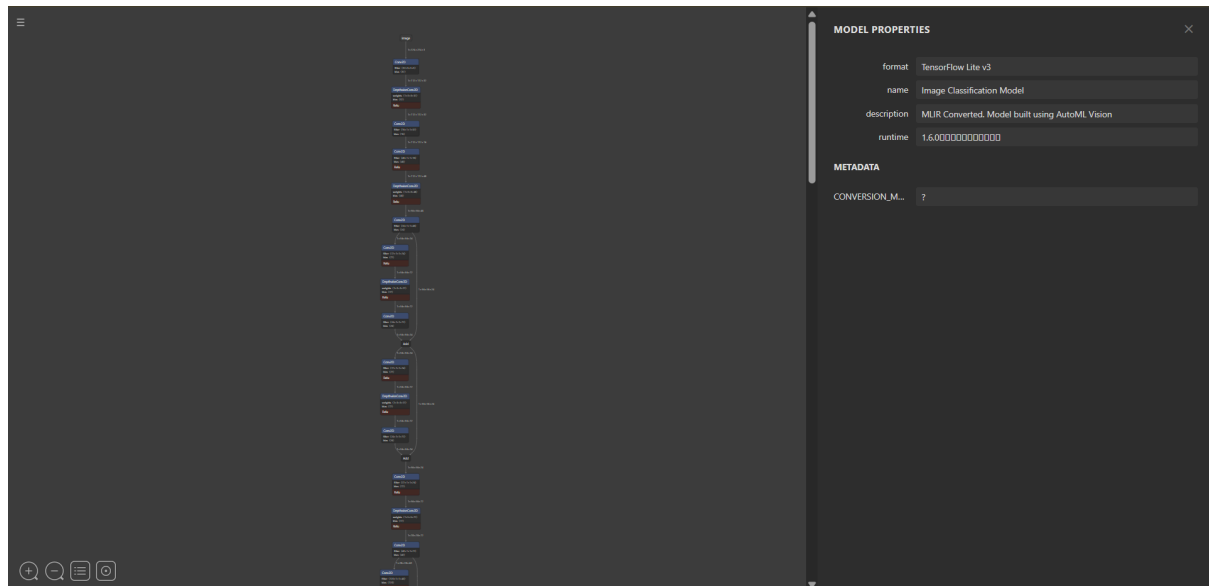


Figure 4.6.3 shows the model analysis on Neutron

To visually verify this structural compression, the exported **.tflite** model topography was analyzed using Neutron, a visualizer for neural network architectures. Inspection of the individual operational nodes confirms the successful execution of the quantization pipeline. The tensor attributes explicitly demonstrate that the convolution layers operate using **int8** precision rather than standard floating-point mathematics, proving the payload is mathematically optimized for edge environments.

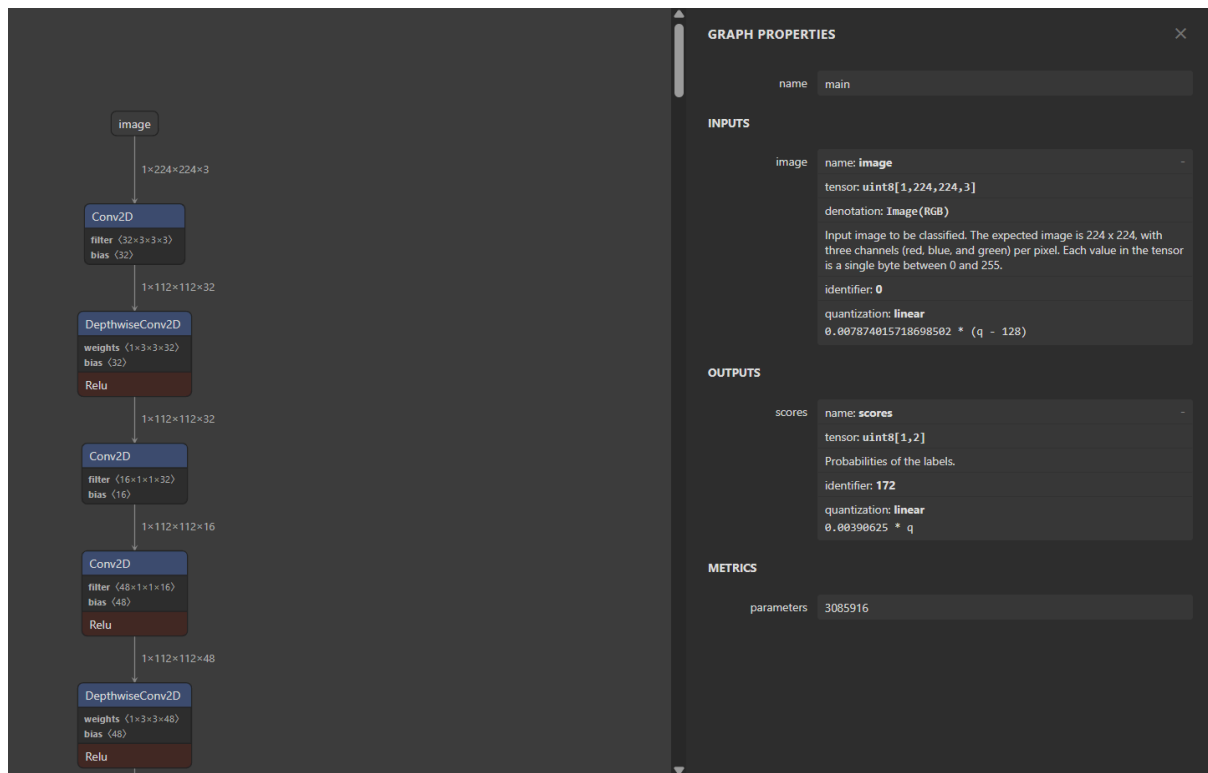


Figure 4.6.4, side panel shows the selected Conv2D node is with tensor `uint8[1,224,224,3]` , suiting for mobile edge model processes

4.6.3 Asynchronous Edge Integration (React Native)

The real-time building-recognition pipeline follows an *Eye-versus-Brain* architecture that exploits the JavaScript Interface (JSI) to bypass the React Native asynchronous bridge, keeping the UI thread free while neural inference executes in the background.

1. Data Ingestion (The "Eye"):

`react-native-vision-camera` delivers zero-copy native camera frames directly via JSI. A frame processor is registered as follows:

```
const frameProcessor = useFrameProcessor(
  (frame) => {
    'worklet';
    // frame: zero-copy reference to the native camera buffer
    // ...inference pipeline runs here on a background thread
  },
  [/* dependencies */]
```



```
);
```

The returned `frameProcessor` is passed to the `<Camera>` component, which invokes it natively on every captured frame without serialisation across the legacy bridge, described in **Section 4.4.1**

2. Thread Delegation:

The `'worklet'` directive delegates execution to a background JavaScript runtime via `react-native-worklets-core`. A `useSharedValue` counter enables lock-free frame throttling:

```
const frameCount = useSharedValue(0);
// Inside worklet:
frameCount.value = frameCount.value + 1;
if (frameCount.value % frameSkip !== 0) return; // default frameSkip = 10
```

Because VisionCamera v4 worklets cannot transfer `jsi::NativeState`, the TFLite model handle is boxed/unboxed across runtime boundaries via `NitroModules`:

```
// JS thread -- box before entering worklet
const boxedModel = model ? NitroModules.box(model) : undefined;

// Worklet thread -- unbox to restore native handle
const model = boxedModel.unbox();
```

3. Hardware-Accelerated Inference (The "Brain"):

The frame is downscaled to the model's input tensor shape, then executed synchronously via `react-native-fast-tflite`, which dispatches to platform accelerators (CoreML / NNAPI / GPU delegate):

```
const resized = resize(frame, {
  scale: { width: 224, height: 224 },
  pixelFormat: 'rgb',
  dataType: 'uint8',
});

const outputs = model.runSync([resized.buffer as ArrayBuffer]);
```

```
// Output: quantised softmax uint8[1,2] → [not-building, building]
const scores = new Uint8Array(outputs[0]);
const conf = Math.max(0, Math.min(1, scores[1] * 0.00390625)); // 1/256
dequant
```

4. State Synchronisation:

The confidence scalar is marshalled back to the React JS thread via a JSI-backed dispatcher, gating the capture sequence:

```
// JS thread -- create bridge callback
const onDetectionResult = useCallback((conf: number) => {
  setConfidence(conf);
  setIsBuildingDetected(conf >= thresholdRef.current); // default threshold =
0.8
}, []);
const runOnJS = Worklets.createRunOnJS(onDetectionResult);

// Worklet thread -- invoke
runOnJS(conf);
```

4.7 Deep Learning Neural Network: Siamese Network

4.7.1 Siamese Network Implementation

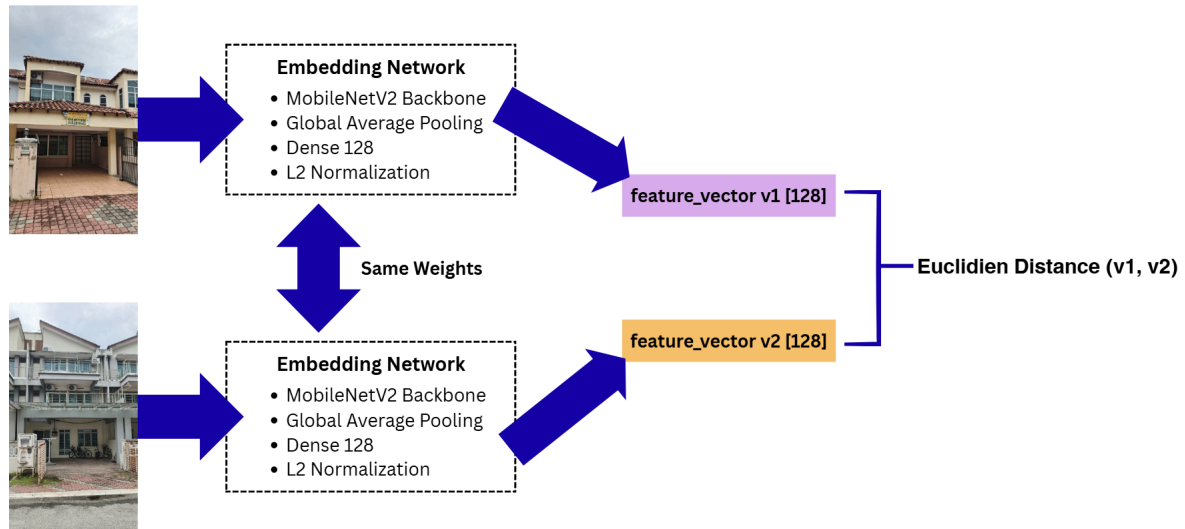


Figure 4.7.1 Architecture pipeline of the implemented Siamese Network verification

To perform building-level verification, the system utilizes a Siamese Convolutional Neural Network architecture as illustrated in **Figure 4.7.1**.

In this project implementation, the architecture design follows a multi-layered approach to extract high-level spatial features from two distinct input images:

- **Input 1:** The reference image of the building from the delivery database.
- **Input 2:** The real-time image captured by the driver's mobile device via the **Dual-Layer Overlay**.

The system extracts compact feature embeddings from both images using a shared convolutional feature extractor (Embedding Network) with same weights and parameters. The distance between these embeddings (feature_vector v1 and v2) is then computed using Euclidean distance. A smaller distance indicates that the two images are more likely to represent the same building, while a larger distance indicates different buildings.

4.7.2 Pair-Based Dataset Design

The training and validation for Siamese Network uses the same housing building datasets used in Section 4.6, with an addition of instead just 1 photo per house, there are 5 images of the same house now: shot from different angles, and alternating brown/white parcels, a human hand simulating a delivery guy's hand, and clean shot.



Figure 4.7.2 shows one of the expanded dataset houses for training the Siamese Network. In the entire dataset, each house consist of 5 images shot from different angle, alternating between various visual noises.

All input images are preprocessed into a consistent format before training. Each image is decoded as a three-channel RGB image, resized to 224×224 pixels, and normalized by scaling pixel values to the range $[0, 1]$. This ensures that all image pairs are compatible with the MobileNetV2-based embedding network.

The dataset is transformed from individual building images into paired samples. Positive pairs are generated by combining two images from the same building folder, while negative pairs are sampled from two different building folders. Each pair is assigned a binary label, where 1 represents the same building and 0 represents different buildings. This pair-based structure allows the Siamese Network to learn visual similarity rather than fixed class labels.

The final datasets contains a total of **1900 pairs** of houses:

- **Total buildings:** 95 distinct terrace houses
- **Images per house:** 5 images
- **Total original images:** 475 images

- **Training houses:** 85 houses (compute to 850 positive pairs and 850 negative pairs)
- **Validation houses:** 10 houses (compute to 100 positive pairs and 100 negative pairs)

4.7.2 Embedding Network Design

```
def build_embedding_network(input_shape=(224, 224, 3),
embedding_dim=EMBEDDING_DIM):
    """MobileNetV2 (frozen) -> GAP -> Dense(128) -> L2-normalized
embedding."""
    backbone = MobileNetV2(
        weights="imagenet",
        include_top=False,
        input_shape=input_shape,
    )
    backbone.trainable = False

    inputs = Input(shape=input_shape, name="embedding_input")
    x = backbone(inputs, training=False)
    x = layers.GlobalAveragePooling2D(name="gap")(x)
    x = layers.Dense(embedding_dim, name="embedding_dense")(x)
    outputs = layers.Lambda(
        lambda t: tf.math.l2_normalize(t, axis=1),
        name="l2_normalize",
        output_shape=(embedding_dim,),
    )(x)

    return Model(inputs, outputs, name="embedding_network")
```

The embedding network is the core feature extractor. According to the architecture illustrated in Figure 4.7.1, it consist of 5 different components layer, each explained below.

MobileNetV2 Backbone

MobileNetV2 is adopted as the convolutional backbone due to its lightweight design and suitability for deployment on resource-constrained environments. Which in our case, the mobile device of the delivery driver. The model is initialized with ImageNet-pretrained weights, allowing it to reuse general visual features such as edges, textures, shapes, and object patterns learned from a large-scale image dataset. The `include_top=False` setting removes the original MobileNetV2 classification layer. This is necessary because the

purpose of this project is not to classify images into ImageNet categories, but to extract meaningful visual features from building images. The backbone is frozen during training so that it acts as a stable feature extractor, reducing training complexity and avoiding overfitting on the limited building image dataset.

Global Average Pooling

After MobileNetV2 backbone extracts convolutional feature maps, a GlobalAveragePooling2D layer is applied. This layer aggregates the spatial dimensions into a single compact feature vector by averaging values across each feature channel. Unlike flattening the entire feature map, global average pooling substantially reduces parameter count, enhancing efficiency and reducing overfitting risk. It also encourages the model to capture the global presence of key visual features, irrespective of their precise spatial locations.

Dense Embedding Layer

A dense layer with 128 units projects the extracted MobileNetV2 features into a compact embedding space. This embedding acts as the learned “building identity representation” used for distance comparison. The 128-dimensional size provides a balance between compactness and representational power. It is small enough for efficient storage and comparison, but still large enough to encode important building identity features.

L2 Normalization

Finally, the embedding vector is L2-normalized so that all embeddings have a consistent scale. This makes Euclidean distance comparison more stable because the distance is influenced mainly by the direction of the feature vector rather than its magnitude. Without normalization, embeddings with larger numerical values could dominate the Euclidean distance calculation even if their visual meaning is not more significant.

4.7.3 The Siamese Twin Architecture

The full Siamese model contains two branches:

- One branch for the reference image
- One branch for the live image

The Siamese architecture uses a twin-tower structure in which both input images are processed by the same embedding network. Since both towers share weights, the system learns a common feature extraction function for both reference and live images. This ensures that the embeddings are comparable within the same feature space.

```
def build_siamese_model(input_shape=(224, 224, 3),
                        embedding_dim=EMBEDDING_DIM):
    """Twin-tower Siamese model using the Functional API."""
    embedding_network = build_embedding_network(input_shape,
                                                embedding_dim)

    reference_image = Input(shape=input_shape, name="reference_image")
    live_image = Input(shape=input_shape, name="live_image")

    # Shared weights: same embedding_network instance called twice.
    ref_embedding = embedding_network(reference_image)
    live_embedding = embedding_network(live_image)

    distance = EuclideanDistance(name="euclidean_distance")(
        [ref_embedding, live_embedding]
    )

    return Model(
        inputs={"reference_image": reference_image, "live_image":
live_image},
        outputs=distance,
        name="siamese_network",
    )
```

4.7.4 Euclidean Distance Layer

```
class EuclideanDistance(layers.Layer):
    """Computes the Euclidean (L2) distance between two embedding
vectors."""
```

```

def call(self, inputs):
    a, b = inputs
    squared_diff = tf.square(a - b)
    sum_squared = tf.reduce_sum(squared_diff, axis=1,
keepdims=True)
    # epsilon guards against sqrt(0) gradients
    return tf.sqrt(tf.maximum(sum_squared,
tf.keras.backend.epsilon()))

def compute_output_shape(self, input_shape):
    return (input_shape[0][0], 1)

```

After both images are converted into 128-dimensional embeddings, the system computes the Euclidean distance between them. Euclidean distance measures how far apart the two embedding vectors are in the learned feature space. A smaller distance indicates that the two embeddings are close together, meaning the two images are likely to belong to the same building. A larger distance indicates that the embeddings are far apart, meaning the images are more likely to represent different buildings.

$$D(\mathbf{x}_1, \mathbf{x}_2) = \sqrt{\sum_{i=1}^n (f_i(\mathbf{x}_1) - f_i(\mathbf{x}_2))^2}$$

- $D(x_1, x_2)$ is the Euclidean distance between the two inputs.
- $f(x_1)$ and $f(x_2)$ represent the embedding vectors generated by the network.
- n is the dimensionality of the embedding space (in this case, $n = 128$).

In the implementation, the distance is calculated by subtracting the two embeddings, squaring the difference, summing the squared values, and applying the square root. A small epsilon value is used to avoid numerical instability when the distance is zero.

4.7.5 Contrastive Loss

```
def contrastive_loss(y_true, y_pred):  
    """  
    Contrastive loss for Siamese networks.  
  
    y_true = 1 → same house (positive pair) → penalise large distance  
    y_true = 0 → different house (negative) → penalise distance < margin  
    """  
    y_true = tf.cast(y_true, y_pred.dtype)  
    positive_term = y_true * tf.square(y_pred)  
    negative_term = (1.0 - y_true) * tf.square(tf.maximum(MARGIN - y_pred, 0.0))  
    return tf.reduce_mean(positive_term + negative_term)
```

Contrastive loss is used to train the Siamese Network. For positive pairs, where both images belong to the same building, the loss penalizes large embedding distances. This pulls same-building embeddings closer together. For negative pairs, where the two images belong to different buildings, the loss penalizes distances smaller than a predefined margin. This pushes different-building embeddings apart until their distance exceeds the margin.

$$\mathcal{L}(y, D) = y \cdot D^2 + (1 - y) \cdot \max(m - D, 0)^2$$

- L is the calculated loss for the current batch.
- y is the ground-truth label ($y = 1$ for a positive pair/same building, $y = 0$ for a negative pair/different buildings).
- D is the Euclidean distance calculated in the previous layer.
- m is the predefined margin (the minimum acceptable distance between different buildings, in this case $m = 1.0$).

Positive pairs are optimized toward zero distance, while negative pairs only contribute loss when their distance is less than the margin. This prevents the model from unnecessarily pushing already well-separated negative pairs farther apart.

4.7.6 Training Configuration

training setup uses:

- **Optimizer:** Adam
- **Learning rate:** 0.001
- **Epochs:** 15
- **Batch size:** 16
- **Loss:** Contrastive loss
- **Backbone:** Frozen MobileNetV2
- **Embedding dimension:** 128
- **Input size:** $224 \times 224 \times 3$

Adaptive Moment Estimation (Adam) gradient-based optimization algorithm used to update network weights after backpropagation computes the gradients. It efficiently determine how to adjust learning rates based on gradients.

The model is trained using the Adam optimizer with a learning rate of 0.001 for 15 epochs and a batch size of 16. Remaining parameters and settings are as described above: The MobileNetV2 backbone is frozen during training, while the dense embedding projection layer learns to map building images into a 128-dimensional normalized feature space. The model is optimized using contrastive loss with a margin of 1.0.

4.7.7 Deployment Design

After training, the embedding sub-network is extracted from the full Siamese model and converted into TensorFlow Lite format. This produces a lightweight model named `building_dna_extractor.tflite`.

```
embedding_network = model.get_layer("embedding_network")
converter =
tf.lite.TFLiteConverter.from_keras_model(embedding_network)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_model = converter.convert()

TFLITE_OUTPUT.write_bytes(tflite_model)
size_kb = TFLITE_OUTPUT.stat().st_size / 1024
```

```
print(f"\nSaved: {TFLITE_OUTPUT.name} ({size_kb:.1f} KB)")
print("Done.")
```

This deployment design is efficient because the system only needs the embedding extractor during inference. During inference, each image is passed through the TensorFlow Lite embedding extractor to obtain a 128-dimensional building descriptor. The Euclidean distance between the reference embedding and the live embedding is then computed. If the distance is small, the pair is considered visually similar, thus resulting in a high probability of they are the same POD image of the same building. If the distance is large, the pair is considered distinct, thus it is not the same building. A threshold can be selected based on validation performance to convert the distance score into a final same-building or different-building decision.

4.7.6 Web User Interface for Evaluation of Siamese Network

The evaluation interface was developed using **Streamlit**, an open-source Python library that transforms data scripts into interactive web apps. It provides an interactive web-based environment for testing and validating the Siamese Network model. The interface supports single image comparison, batch testing, automatic pair generation, and full model validation using a validation folder and pair CSV file.

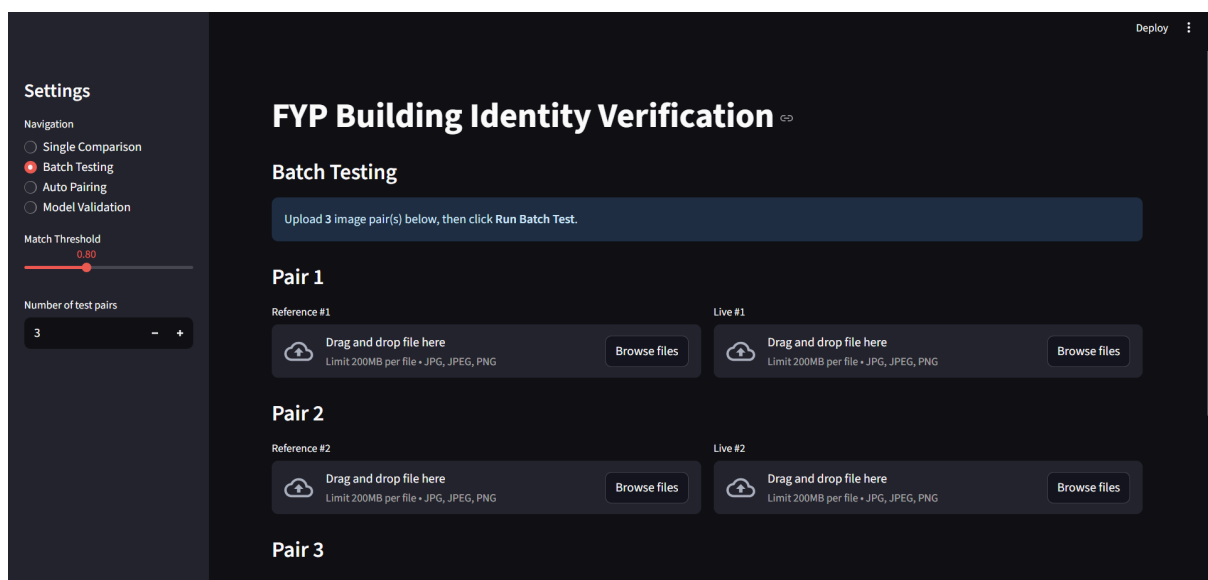


Figure 4.7.6 The User Interface design before CSS for the Siamese Network evaluations

The web interface allows users to interactively adjust the matching threshold, observe model decisions, and analyze validation performance using standard biometric verification metrics:

The interface estimates the Equal Error Rate by sweeping multiple threshold values. The threshold where FAR and FRR are closest is reported as the optimal EER threshold. The interface incorporates a confusion matrix that summarizes the model's binary classification performance across four key outcomes:

- **True Positives**, where same-building pairs are correctly identified
- **True Negatives**, where different-building pairs are correctly rejected
- **False Positives**, representing erroneous acceptance of different-building pairs
- **False Negatives**, indicating missed same-building matches.

This matrix enables precise quantification of the model's discrimination capability between visually similar and dissimilar building images.

A **distance distribution histogram** visualizes the empirical distribution of Euclidean distances for positive (same-building) and negative (different-building) pairs, highlighting the desired bimodal separation where intra-class distances cluster near zero and inter-class distances are larger. A red dashed line denotes the active threshold, aiding in the intuitive assessment of classification boundary efficacy.

The **FAR vs. FRR curve** plots the False Acceptance Rate (fraction of different-building pairs incorrectly matched) against the False Rejection Rate (fraction of same-building pairs incorrectly rejected) over a spectrum of threshold values, akin to a receiver operating characteristic analysis tailored to verification tasks. A green dashed line marks the Equal Error Rate threshold, optimizing the balance between error types.

Additionally, a **per-pair results table** furnishes granular forensic analysis, enumerating for each validation instance: pair identifier, computed Euclidean distance, ground-truth label, threshold-derived predicted label, and binary correctness indicator.

CHAPTER 5 System Implementation

5.1 Hardware Setup

The hardware involved in this project is a laptop and android mobile device. A laptop is issued for the development of the application, specifically code writing and database initial settings. A mobile device is used for testing and showing the changes with respect to the code logic in real-time, thus streamlining the debugging workflow. The mobile device should have a geolocation function, and a built in working camera system, which both will be accessed by the application. Both the laptop and mobile device should have internet connectivity in order to connect to the same network to communicate. This restriction is required in the development phase of the system and can be redundant once the system is deployed for production.

<i>Component</i>	<i>Specifications</i>
<i>Model</i>	<i>ASUS Vivobook</i>
<i>Processor</i>	<i>11th Gen Intel Core i5-11300H</i>
<i>Operating System</i>	<i>Windows 11 Home Single Language 64-bit</i>
<i>Graphic</i>	<i>Intel Iris Xe Graphics</i>
<i>Memory</i>	<i>8GB DDR4 RAM</i>
<i>Storage</i>	<i>256GB NVMe SSD</i>
<i>Connectivity</i>	<i>Intel Wi-Fi 6 AX201 160MHz</i>

Table 5.1 Specifications of the Laptop as development workstation

Description	Specifications
Model	OPPO Find X8
Device Type	Android

OS Version	Android version 16
Device Sensors	GNSS/GPS Receiver, Rear Camera (50MP+50MP+50MP)
Connectivity	4G/LTE or Wi-Fi
Storage	512 GB
RAM	16.0 GB

Table 5.2 Specifications of the mobile device for development and debugging

5.2 Software Setup

5.2.1 Integrated Development Environment and Language

The software included in this project includes WebStorm as the Integrated Development Environment (IDE) by JetBrains. It is used to facilitate the development of JavaScript and TypeScript as the main programming languages of the mobile application. Node.js will be used as the JavaScript runtime environment. React Native framework is used to achieve cross-platform capabilities. The application will also be developed under Expo Application Services (EAS) environment, which is a deeply integrated cloud services for Expo and React Native apps. NativeWind is used to enhance the efficiency of designing Cascading Style Sheets (CSS) styling for the UI. The application will be utilizing Supabase for Backend-as-a-Service, specifically as a Database and Storage system. Git and Github will be used, to allow version control and streamline the development.

Category	Software / Tool	Version
Development	WebStorm (JetBrains)	2025.2.4
	Node.js	24.11.1
	ESLint	9.25.0
	Prettier	3.8.1

Programming Language	TypeScript	5.9.2
	JavaScript	ES6+
Frameworks	React Native	0.81.5
	React	19.1.0
	Expo SDK	54.0.25
	Expo Router	6.0.15
	NativeWind	4.0.1
	TailwindCSS	3.4.1
Development Service	Expo Application Services (EAS) CLI	≥ 16.28.0
Backend & Data Storage	Supabase (supabase-js)	2.98.0
	Deno (Edge Functions runtime)	2.x
Machine Learning	TensorFlow Lite (react-native-fast-tflite)	3.0.0
Version Control	Git	Latest
	GitHub	—

Table 5.2.1 Specifications of the Software and Development Tools

5.3 Development Environment Setup

5.3.1 Expo Application Service

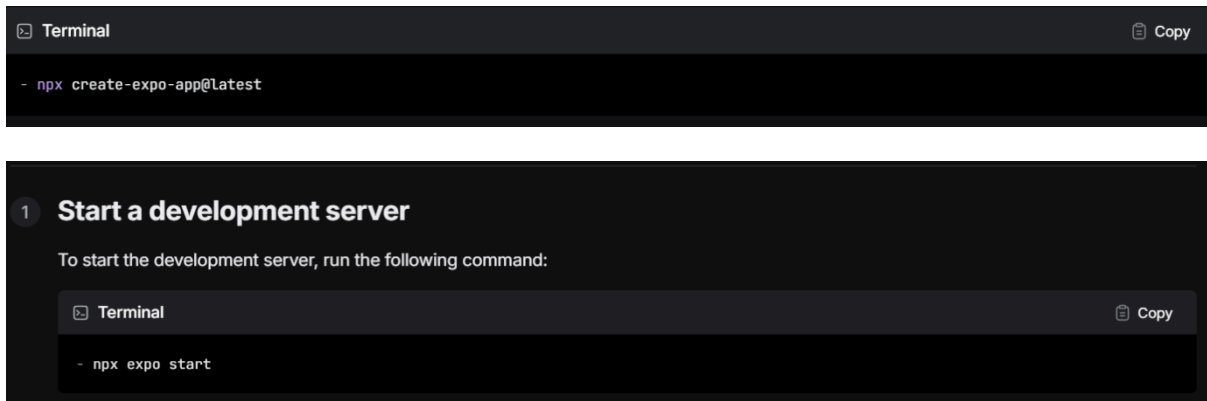


Figure 5.3.1 Command Line interface that create initial Expo project repository

The figure above shows the command line prompt that is needed to create the initial Expo project repository. In the development phase, the connection and simulation of the app runs on the mobile device via Expo Go, an application installed on the mobile device to allow real-time debugging and testing. After running the command, Expo will generate a Quick Response (QR) Code that once scanned by Expo Go will generate and run an initial demo app interface to showcase its basic capabilities.

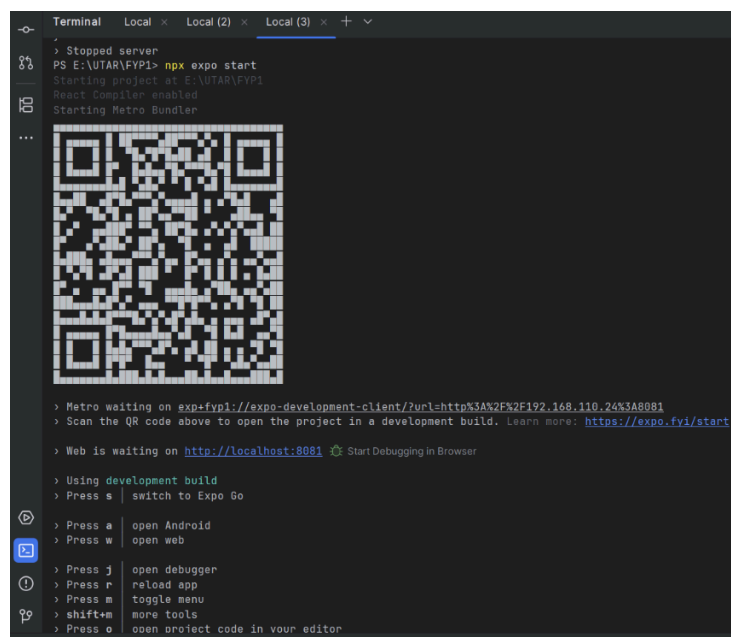


Figure 5.3.2 QR Code generated by EAS for direct testing on physical mobile device

However, the Expo Go environment is only suitable for initial testing of an application. Since the system will be using 3rd party libraries that is not part of the standard Expo Standard Development Kit (SDK), and handling on underlying OS for complex hardware management

(eg, high-precision GPS, custom EXIF handling), the EAS environment is more suitable in order to allow development of customized native code logics. Expo Cloud Dashboard is needed to be created in order to use EAS. The figure below showcases the dashboard that allows developers to view insights, and control the system.

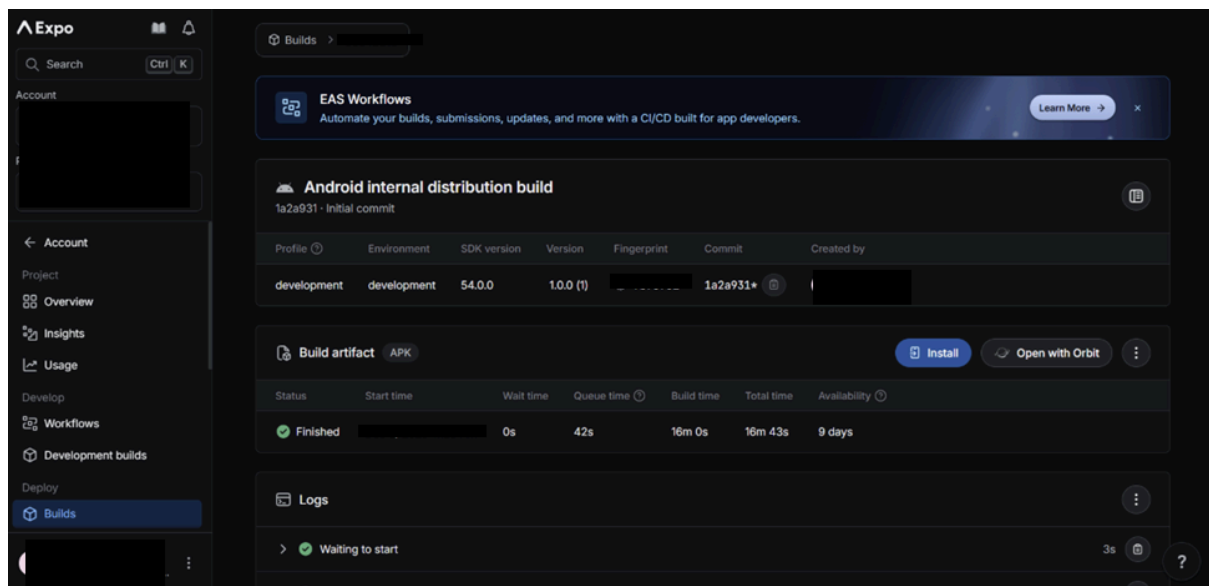
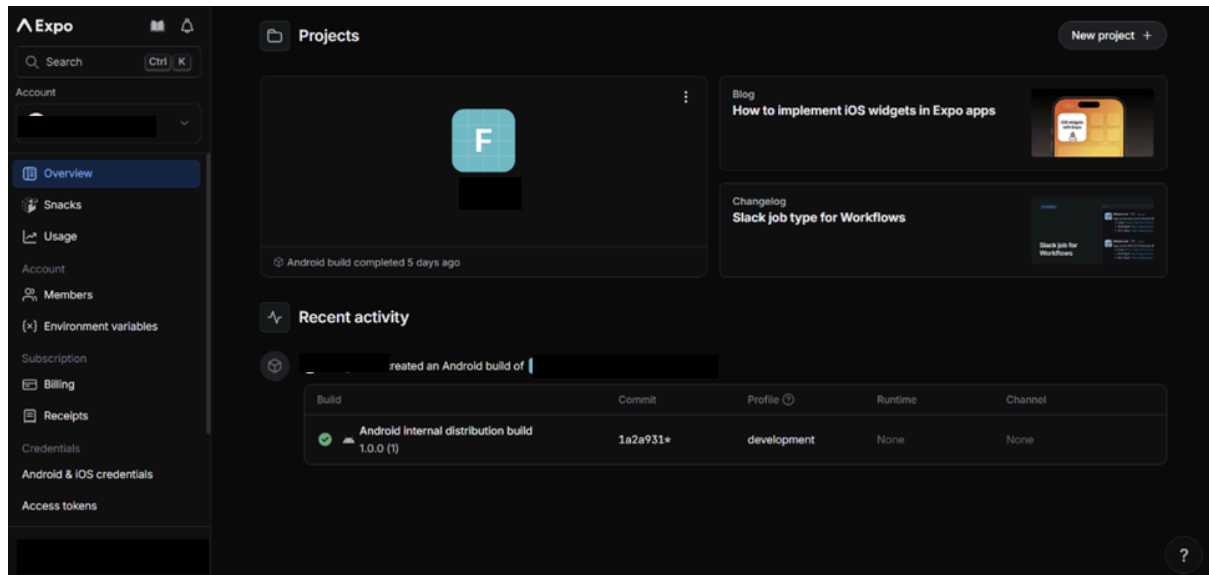


Figure 5.3.3, Figure 5.3.4 both shows the dashboard UI of EAS

5.3.2 Dependencies on Project Repository

A couple of config files and dependencies are needed to be created in the project repository to support the development cycle. Specific native dependencies were integrated into the build pipeline. Table 5.3.2 below details the justification for the package selection for this initial phase.

Module Category	Package Name	Version	Justification
UI Styling	<code>nativewind</code>	4.0.1	Enables TailwindCSS utility classes in React Native components
	<code>tailwindcss</code>	3.4.1	Utility-first CSS framework powering NativeWind's styling engine
Geolocation	<code>expo-location</code>	18.0.7	Provides GPS coordinate acquisition with configurable accuracy
Hardware Access	<code>react-native-vision-camera</code>	4.7.3	High-performance camera module with frame processor support
	<code>vision-camera-resize-plugin</code>	3.2.0	Resizes camera frames for efficient ML model input
	<code>expo-media-library</code>	18.2.1	Read/write access to the device photo library for saving captures
	<code>expo-battery</code>	10.0.8	Reads device battery level for metadata enrichment

EXIF Metadata	<code>@lodev09/react-native-exif</code>	0.2.7	Native EXIF read/write for on-device image metadata injection
	<code>exiftool-vendored</code>	33.5.0	Server-side EXIF parsing and validation via ExifTool binary

Table 5.3.2 Core Dependency Table

Babel is used in this project as the JavaScript compiler, configured through `babel-preset-expo` as the base preset to transpile modern TypeScript and JSX syntax into code compatible with the React Native runtime. It is further extended with the `nativewind/babel` preset, which transforms TailwindCSS utility classes into React Native-compatible styles at compile time by setting `jsxImportSource` to `nativewind`. Two additional plugins are registered: `react-native-worklets-core/plugin`, which enables synchronous JavaScript worklet execution on the camera frame processor thread, and `react-native-reanimated/plugin` (placed last as required), which transforms animation worklet functions to run on the native UI thread for performant gesture and animation handling.

5.4 System Operations

5.4.1 Dual-Layer Overlay and Barcode OCR for Structure Capture

To address the problem of unstructured POD image data, the application enforces a “Dual-Layer” Overlay UI Composition rule. When users navigate to the camera capturing page, the interface is programmatically divided into top and bottom sections, where each zone serves a different purpose in contextualizing the proof image taken.

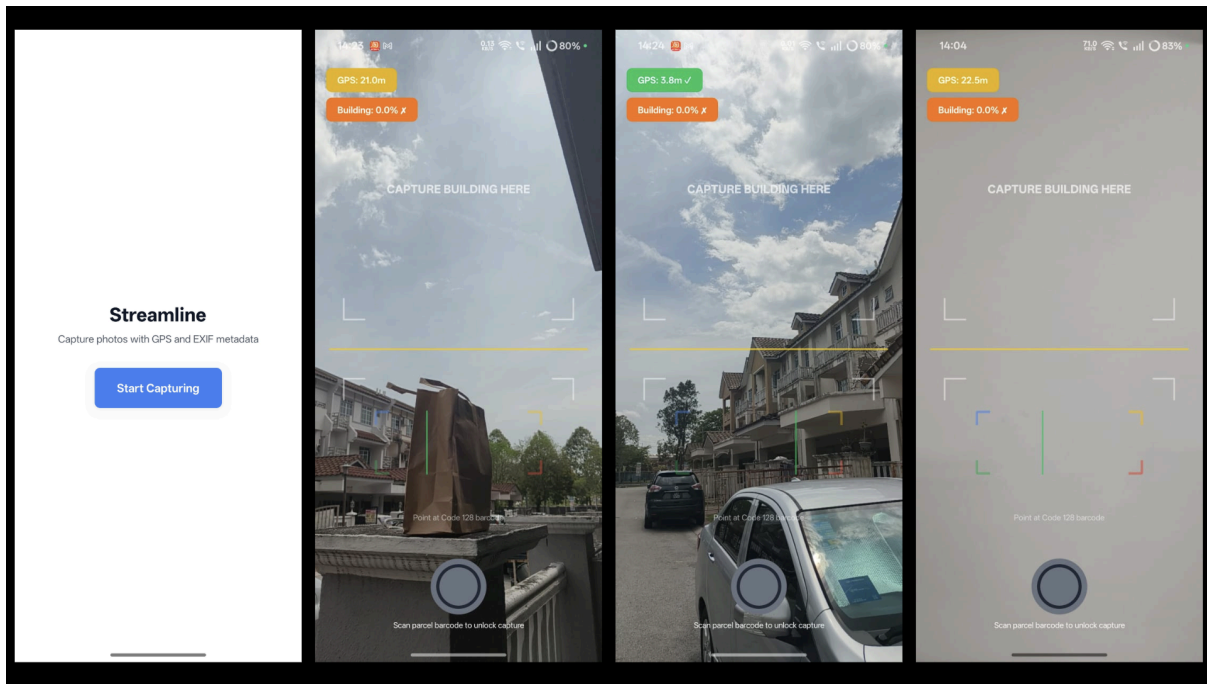


Figure 5.4.1 The in-app Screenshots of the Camera View under different backgrounds

As shown in the application screenshot above, the upper 50% frame, called the Context Zone, is designed to overlay the Location Background of the delivery destination to provide contextualized information by capturing building facades or landmarks. The bottom 50% frame, called the Parcel Zone is designed to present the physical parcel information such as labelling, barcode, and condition of the package.

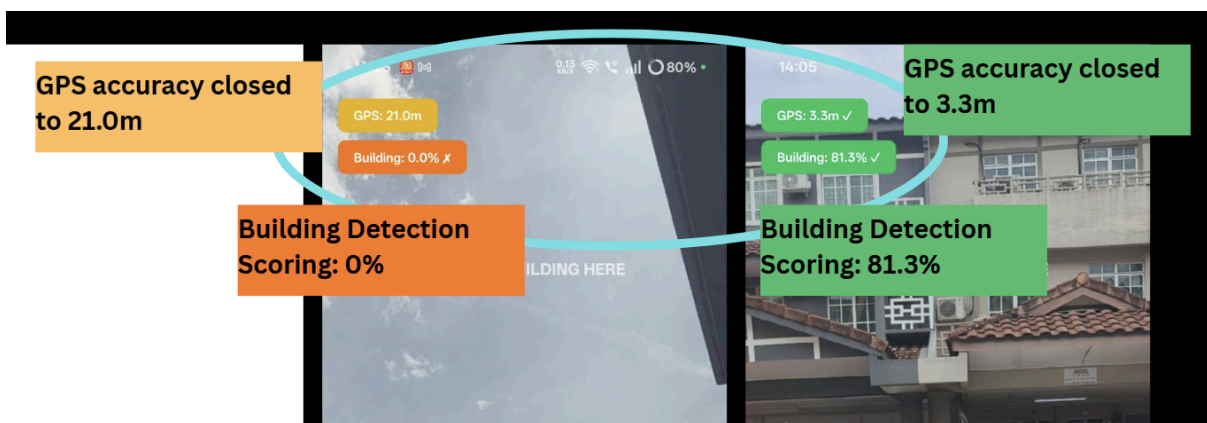


Figure 5.4.2 Labels on-screen for GPS accuracy and Building Detection confidence score

There is two labels on top, one being the constant GPS Accuracy Value and another being the Building Recognition Confidence Scoring. This architectural choice facilitates the simultaneous capture of critical environmental and package-specific data, enhancing the evidentiary value of each POD image.

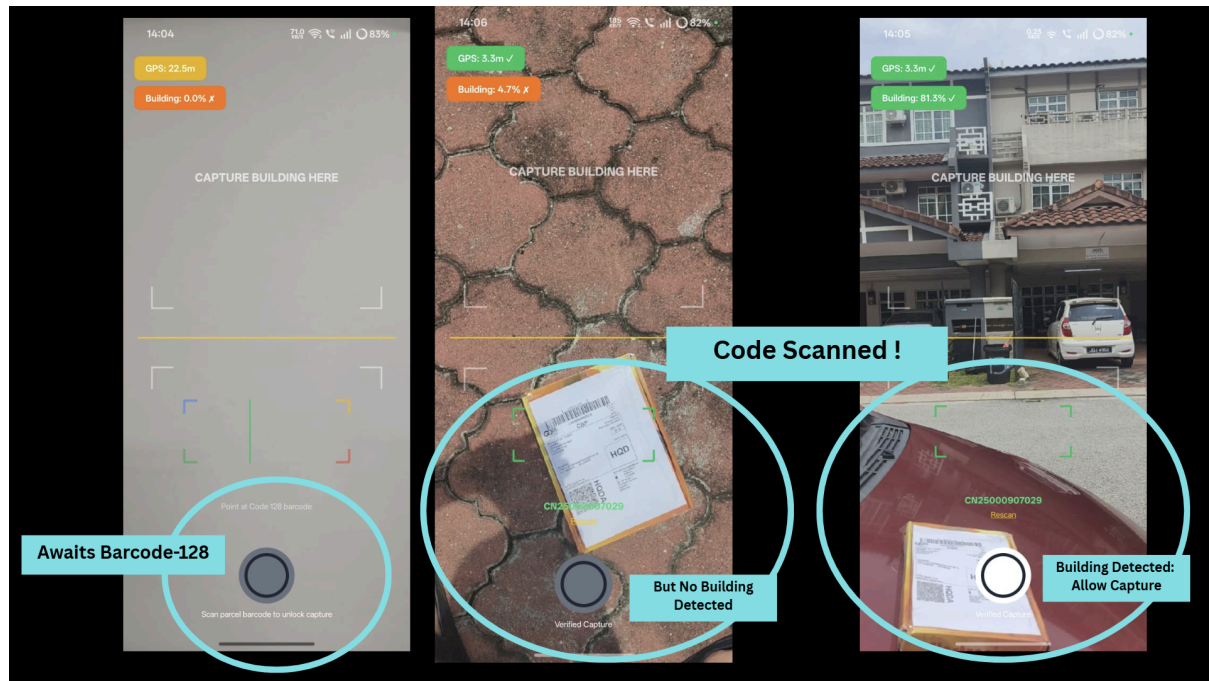


Figure 5.4.3 The logical conditions for enabling capture mode

The bottom layer inherit a barcode scanning animation which is described in detailed in Chapter 4 and is awaiting barcode-128 formats to be scanned. Since there is no barcodes detected, the capture button remains disabled. Upon successful barcode detection and subsequent validation, the capture button dynamically activates, allowing the user to proceed with image acquisition.

5.4.2 Implementation of Geolocation Gating Algorithm

Geolocation Gating Algorithm is a core adaptive mechanism that evaluates the signal strength of the GNSS telemetry stream operated in real-time. It uses the expo-location library, which provides a *Location.watchPositionAsync(options, callback, errorHandler)* function to apply location watching. The model registers a continuous location listener with the device OS location services to evaluate sensor inputs (fuses GNSS/GPS satellites, cell towers, Wi-Fi SSIDs, barometer, magnetometer, accelerometer, and gyroscope) continuously and emits new fixes as they become available.

This section describes the behavioral scenarios that utilized the *latestCoordinateRef* variable derived from above. The system successfully handles three distinct operational states, and will present the respective color code (Green, Red, Yellow) on the UI. A detailed decision table is shown in Table 4.5.1

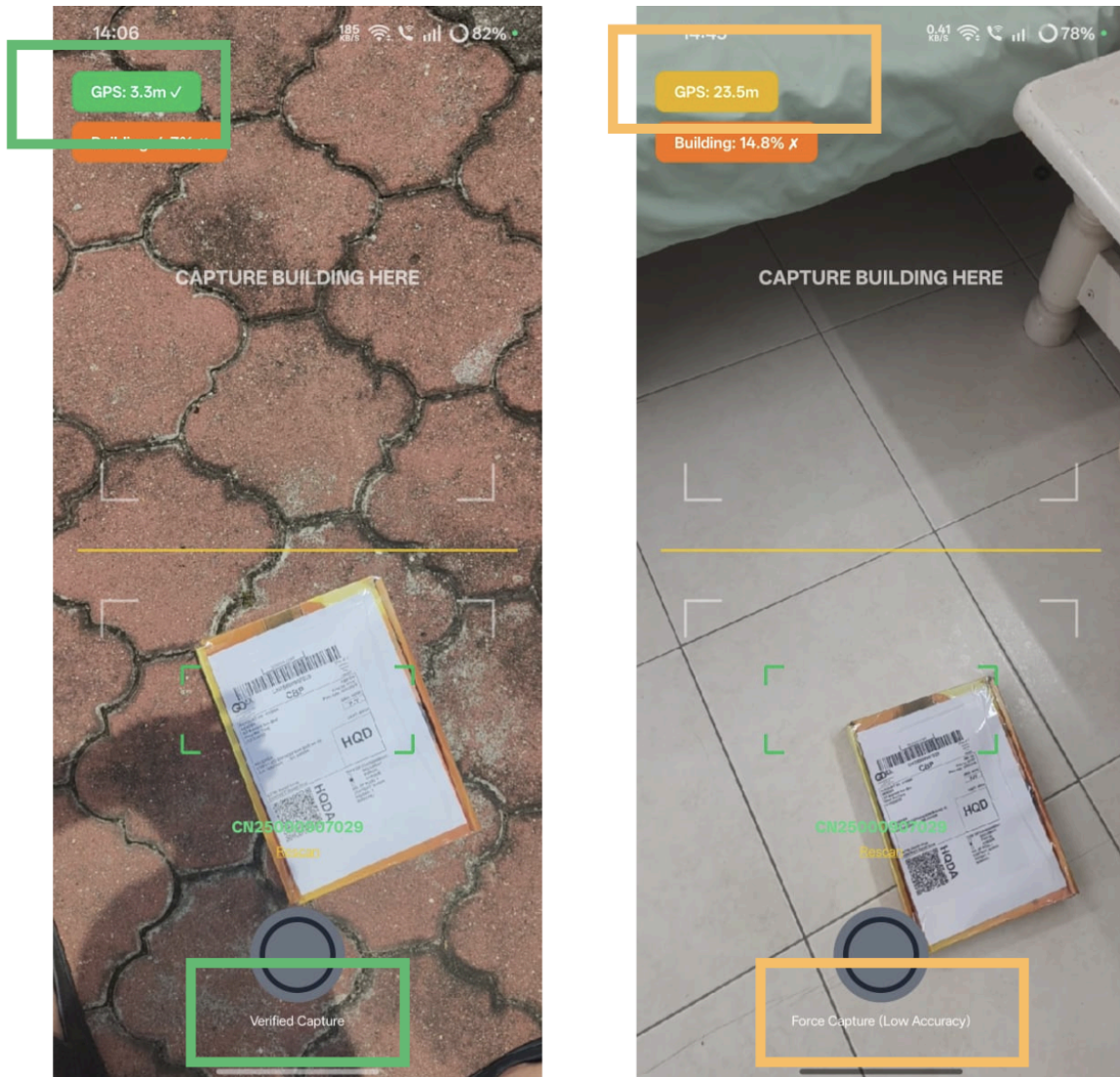


Figure 5.4.4 illustrates the visual representation of these states within the application's user interface, mapping directly to the decision logic outlined in the aforementioned table.

Error Handling

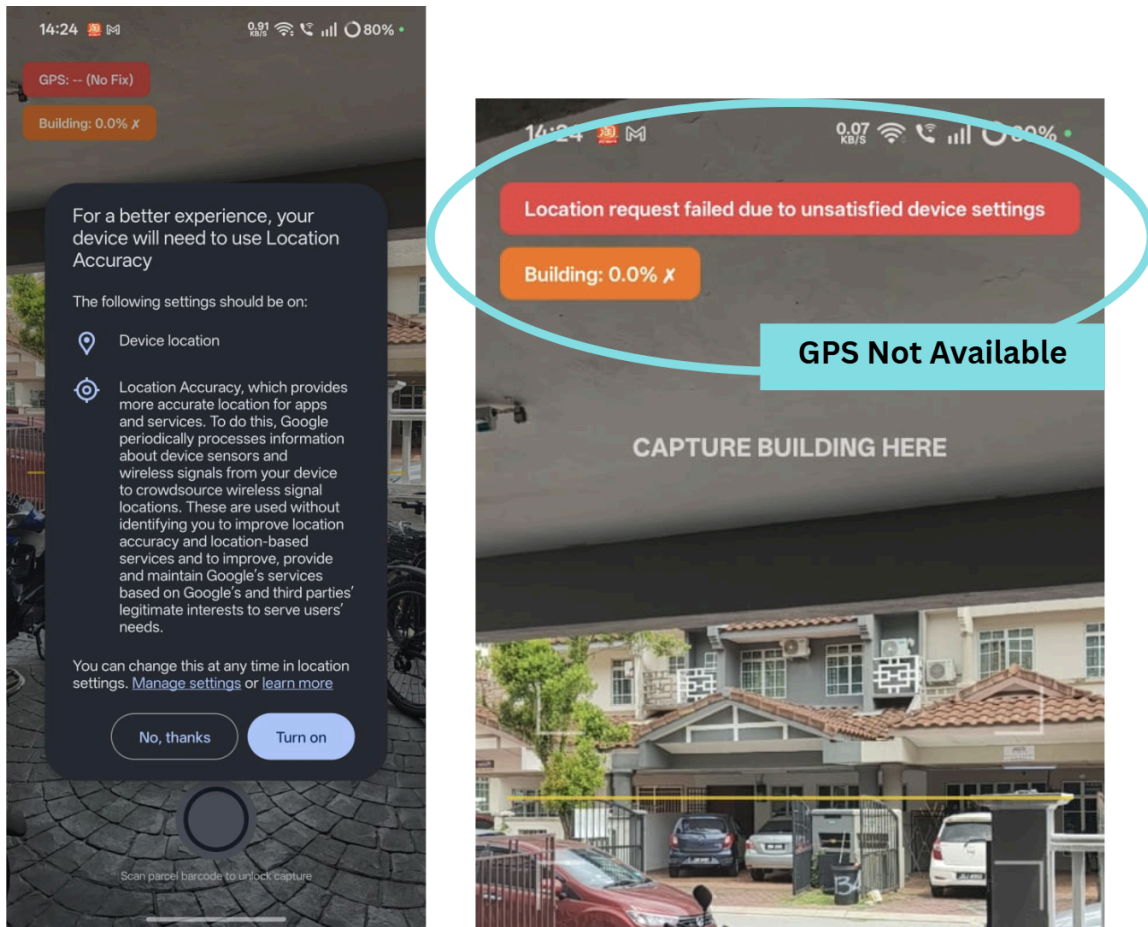


Figure 5.4.5 illustrates the error fallback the app is going to handle if GPS is not available

To maintain system robustness, the application incorporates sophisticated error handling mechanisms, such as gracefully degrading GPS accuracy to network-based localization if satellite signals are weak, or prompting the user for manual location confirmation [34].

5.4.3 Real-Time ML Edge Building Detection

This feature leverages on-device machine learning models to identify building structures within the camera's field of view, providing real-time contextual awareness during image capture. The implementation utilizes a specialized neural network architecture, optimized for efficient inference on mobile platforms, to perform object detection and classification of architectural elements. The development details are outlined in Chapter 4:

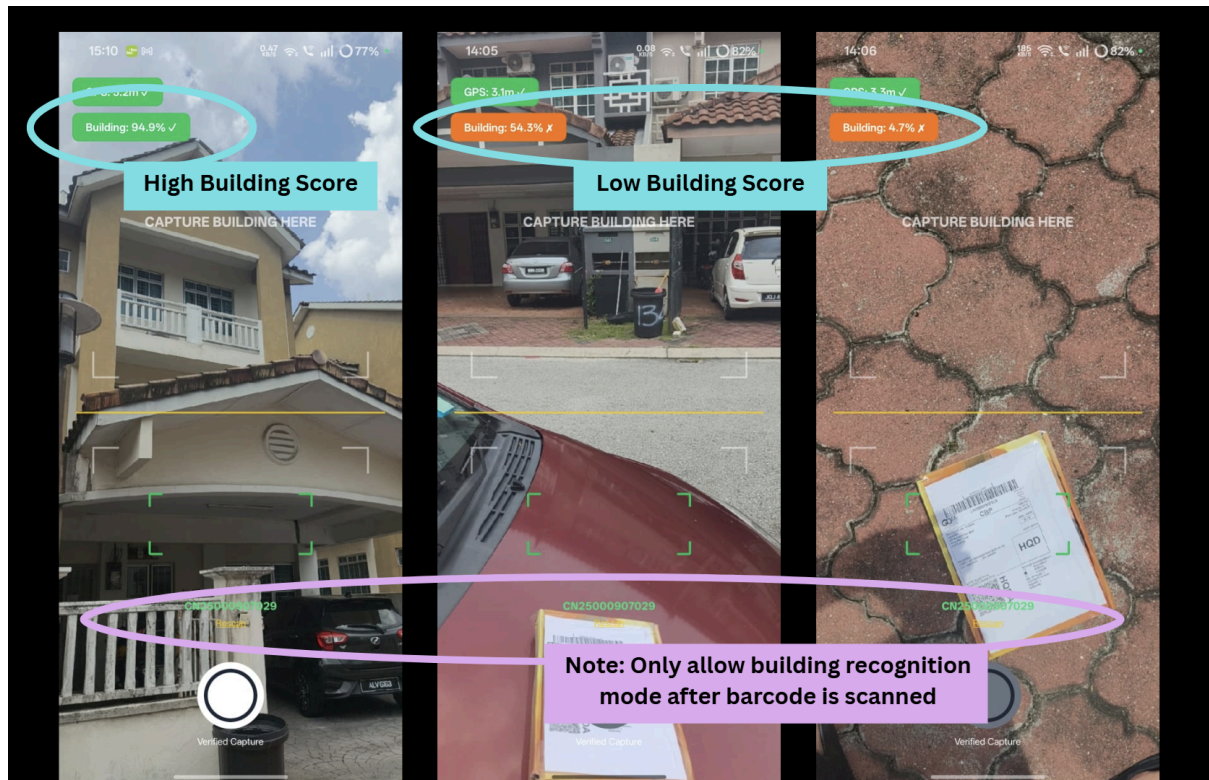


Figure 5.4.6 Building Recognition in Action

The Figure above showcase different confidence scores for detected building structures, allowing the system to quantify the certainty of its real-time building recognition. This continuous assessment of recognition confidence enables the system to refine its contextual data and potentially trigger re-captures or manual verification if confidence levels fall below a predefined threshold, thereby improving the overall reliability of the POD images.

5.4.4 Injecting EXIF Metadata into the Captured Image

Upon image capture, crucial information such as geolocation coordinates (latitude, longitude, altitude), timestamp, and device-specific parameters are programmatically embedded into the image's EXIF metadata fields.

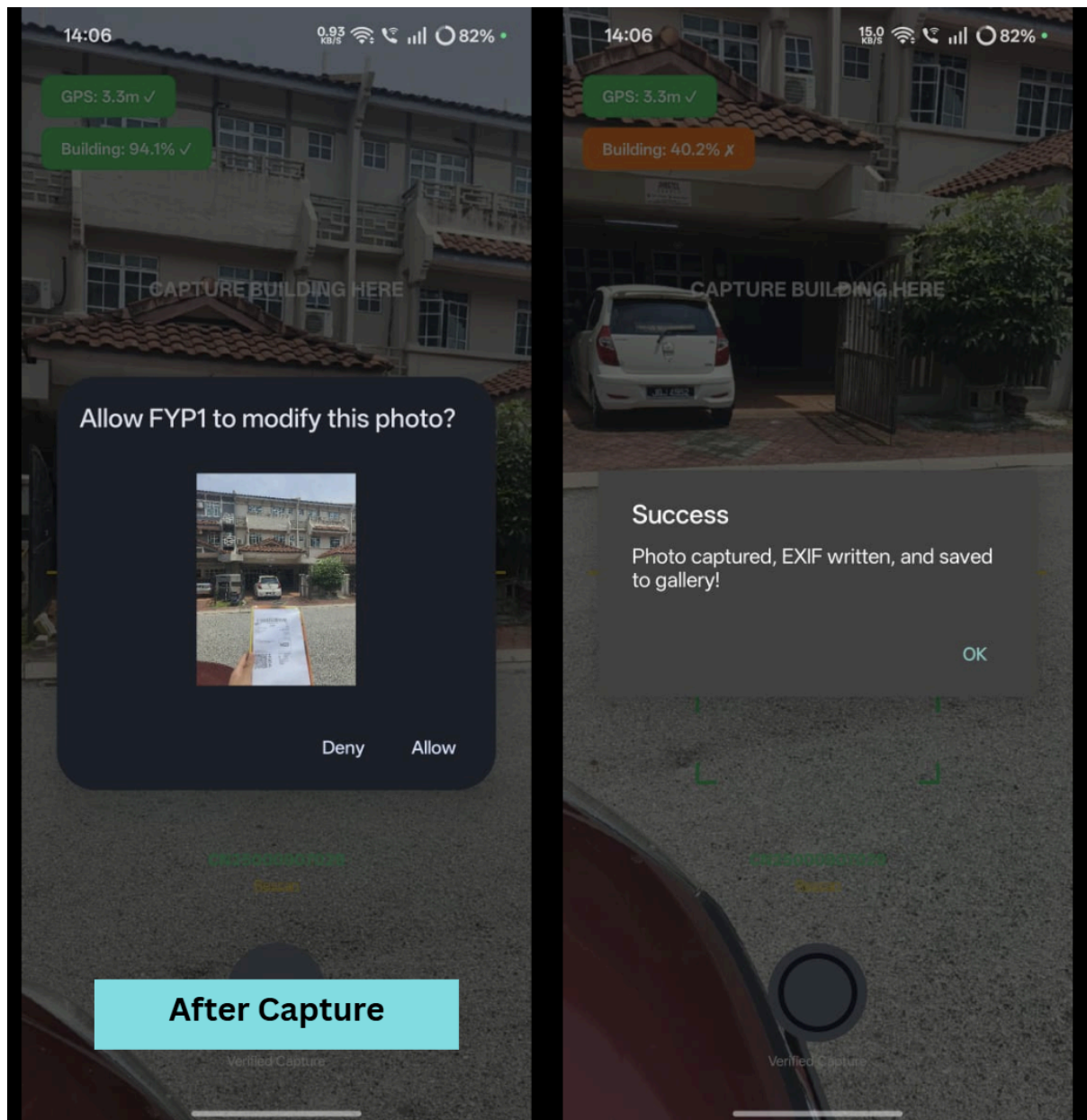


Figure 5.4.7 EXIF Metadata injection in Action

A custom utility file, `exif-processor.ts` is created to handle the injection of EXIF metadata into the raw image captured. A third party library `@lodev09/react-native-exify` is used to burn the EXIF metadata via a function `writeAsync()`. The detailed implementation are described in Chapter 4.

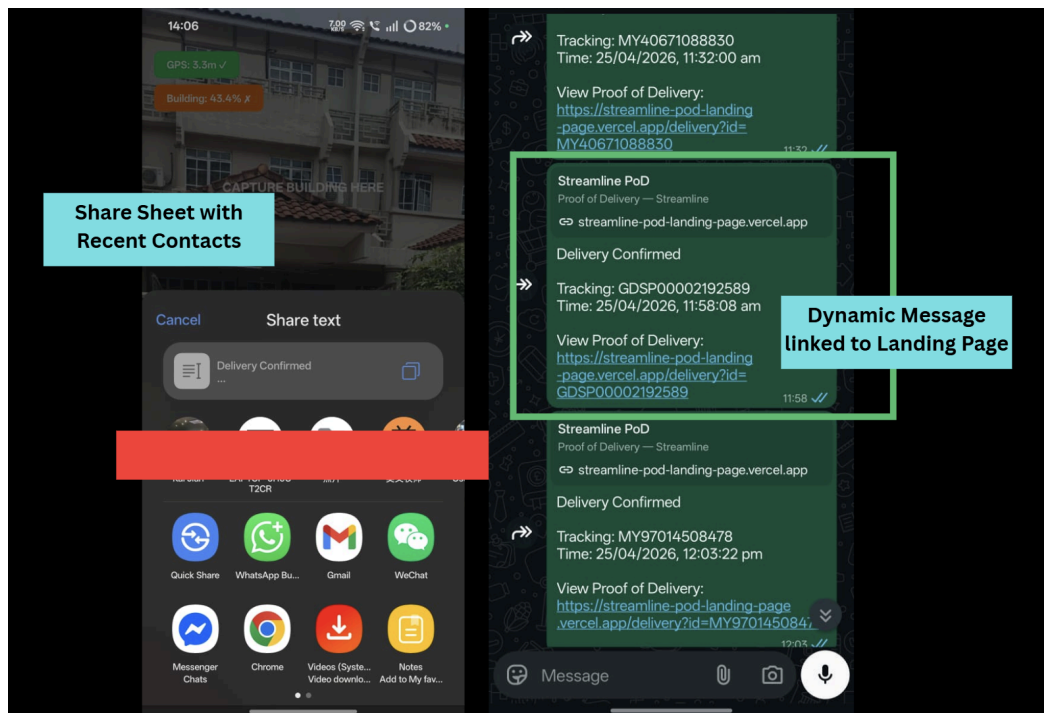


Figure 5.4.8 Share Sheet in Action, sending delivery notifications and POD Landing Page URL to recipients via WhatsApp

After capturing, there is also a Share Sheet feature that allows delivery drivers to disseminate the verified POD image along with its embedded metadata through various communication channels. There will be a auto-generated delivery report that bundles the image and metadata, facilitating seamless integration with existing logistics platforms.

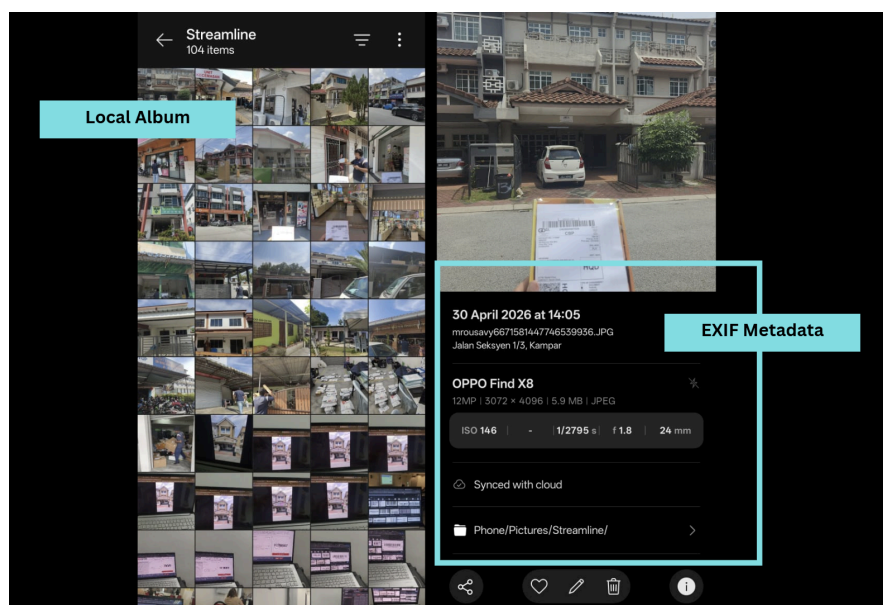


Figure 5.4.9 Local Storage of the POD image are intacted with EXIF Metadata

To address the challenge of network instability in delivery environments, the application implements an Local Storage Architecture. The system utilize expo-media-library to get access to the device local gallery, and saves the image (that has injected EXIF metadata) from the cache to the device's secure document directory. Results are as shown in Figure Above.

5.4.5 POD Landing Page

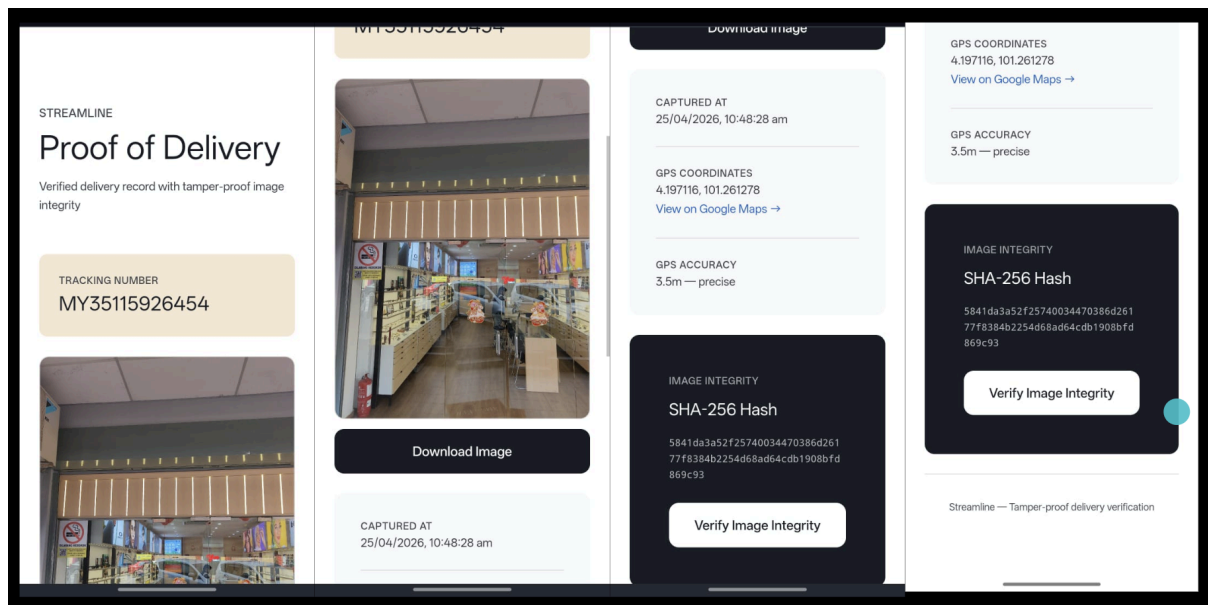


Figure 5.4.10 Screenshot of the POD Landing Page

Figure above depicts the Proof of Delivery landing page, which serves as the central hub for managing and reviewing captured delivery evidence.

Parcel recipients could view their delivery details, including the POD image and its associated metadata, ensuring transparency and accountability in the delivery process. A download button is provided to allow the recipient to save the complete POD report for their records.

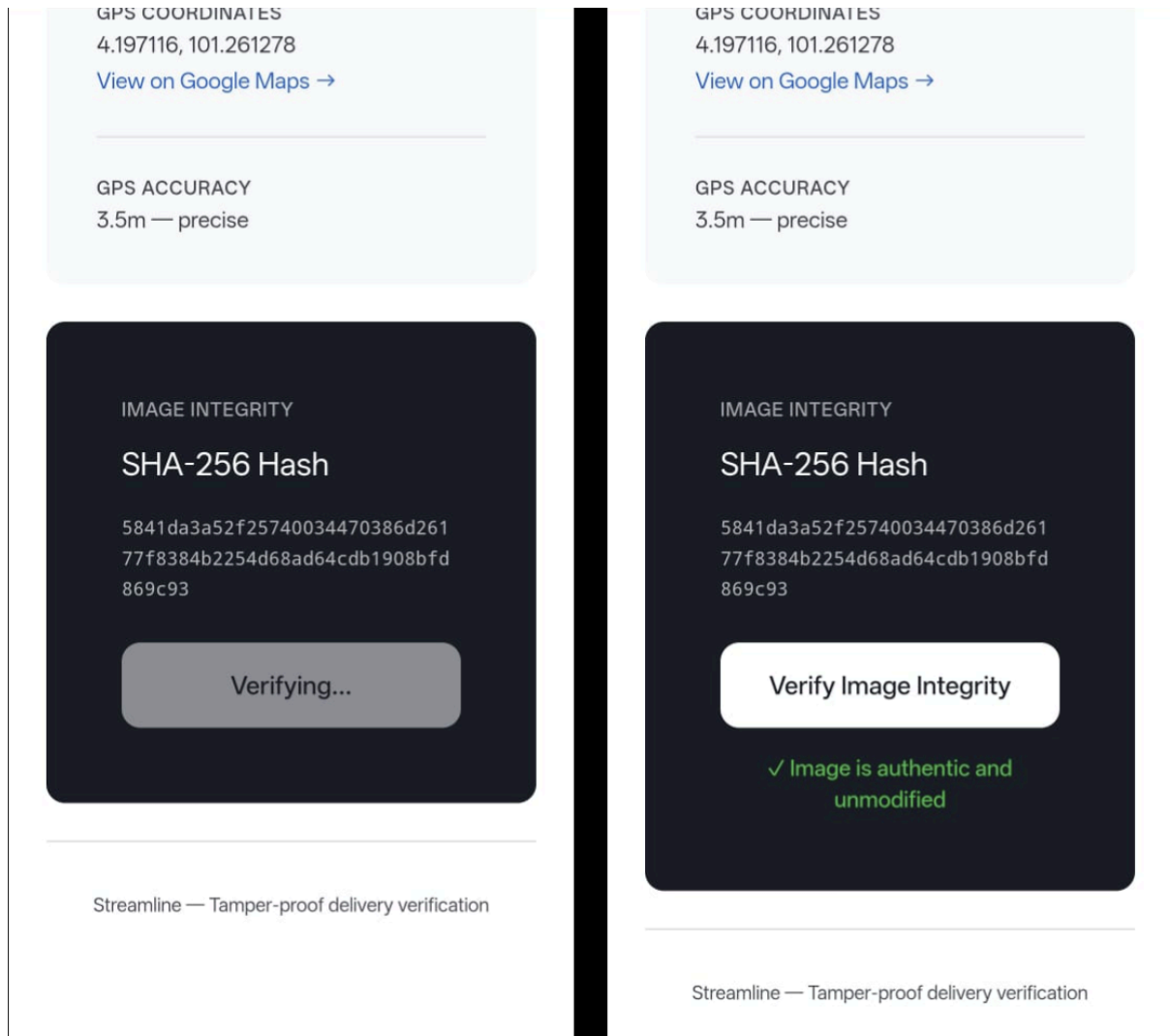


Figure 5.4.11 Hasher Checking of the SHA-256 done on image EXIF

A Hashing check mechanism is implemented to validate if the SHA-256 Hash of the Image is consistent with the initially recorded hash, thereby ensuring the image's integrity and immutability from alteration [35].

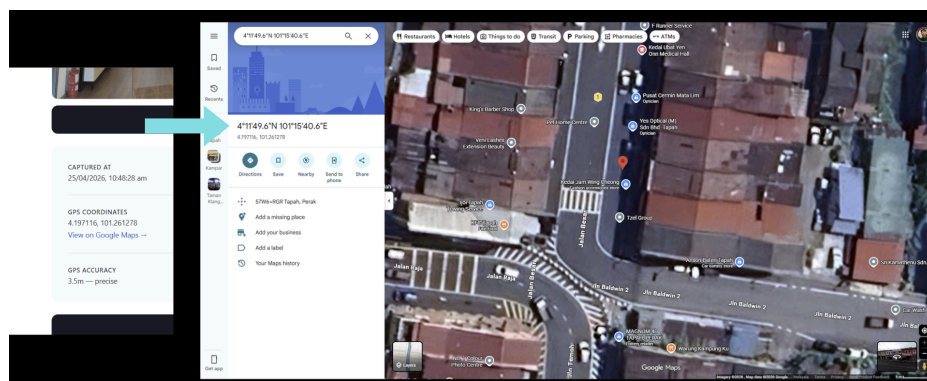


Figure 5.4.12 Clicking on the coordinates directs users to Google Maps

There is also a link to direct users to view the EXIF Images injected geolocation on Google Maps.

5.4.6 Validate Temporal Images with Siamese Network

This advanced validation mechanism employs a Siamese Network architecture to ascertain the temporal consistency and authenticity of sequential images, particularly crucial for detecting manipulations or unauthorized alterations across different capture instances.

There is currently 4 navigation tabs implemented. Namely:

1. Single Comparison

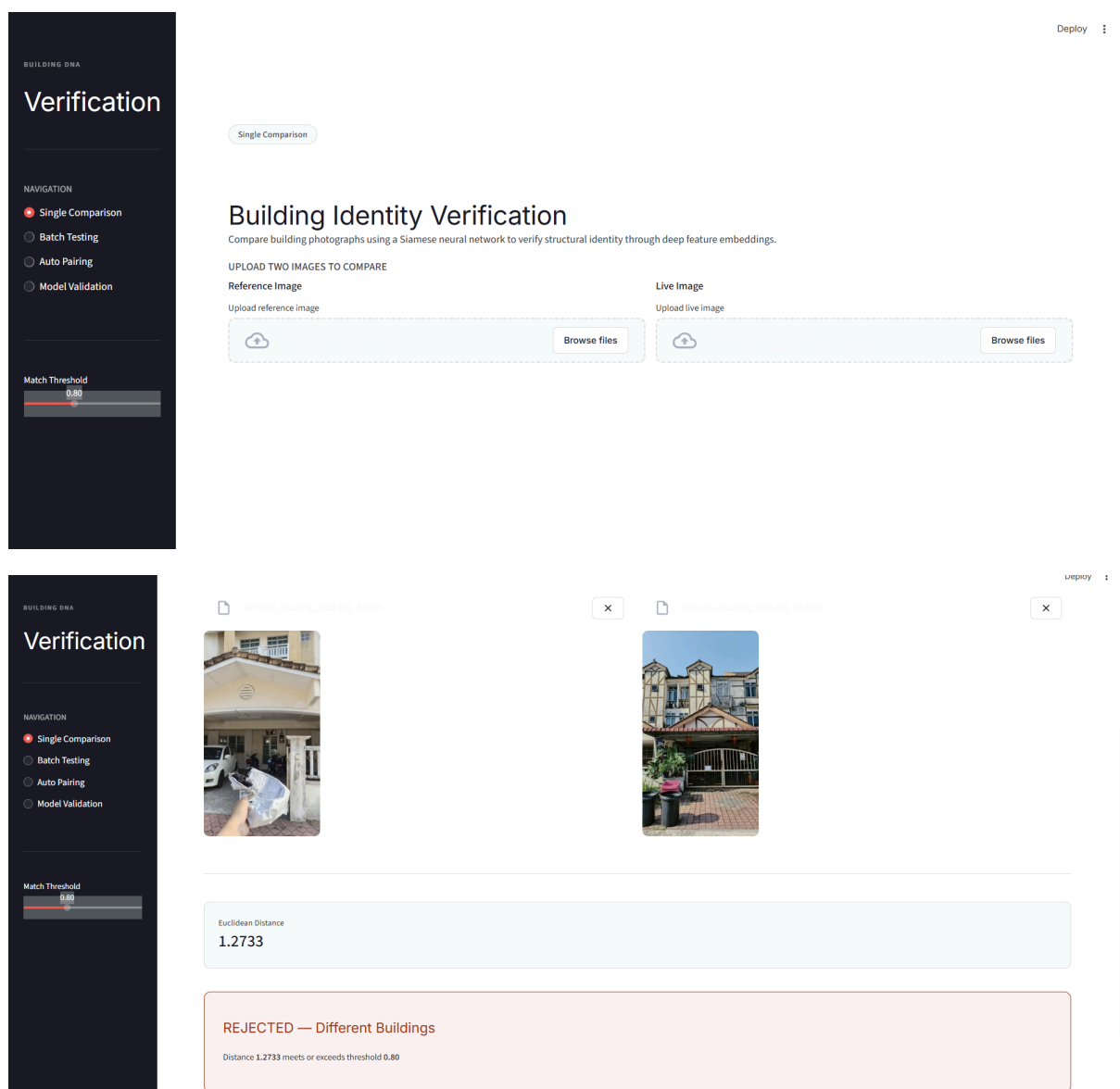


Figure 5.4.13, 5.4.14 Showcase the in-app screenshot of the Single Comparison feature

2. Batch Testing

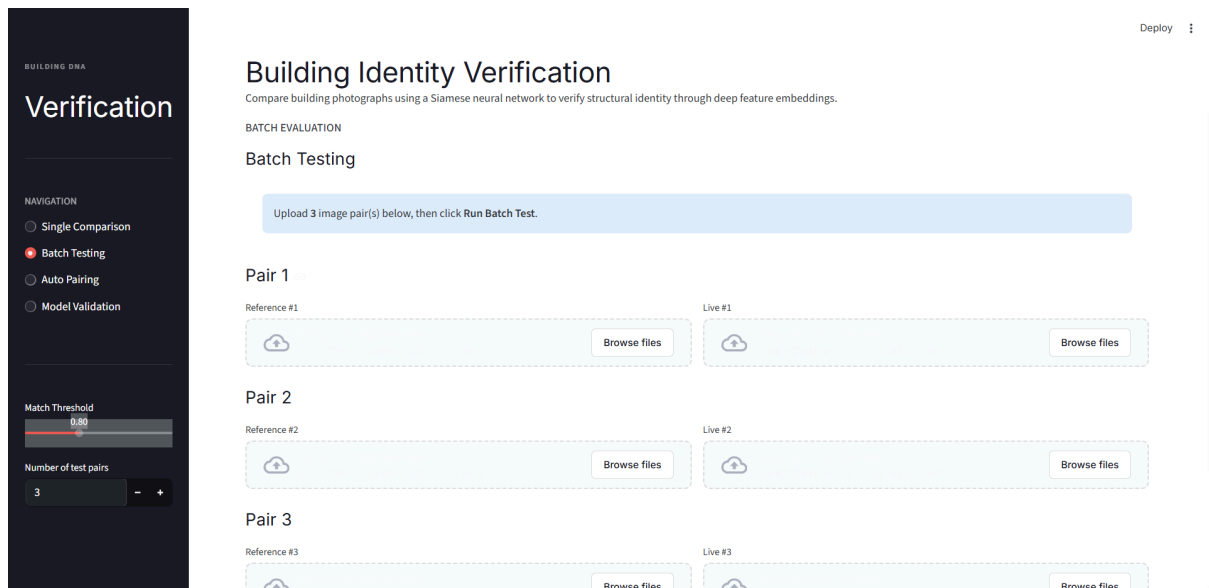
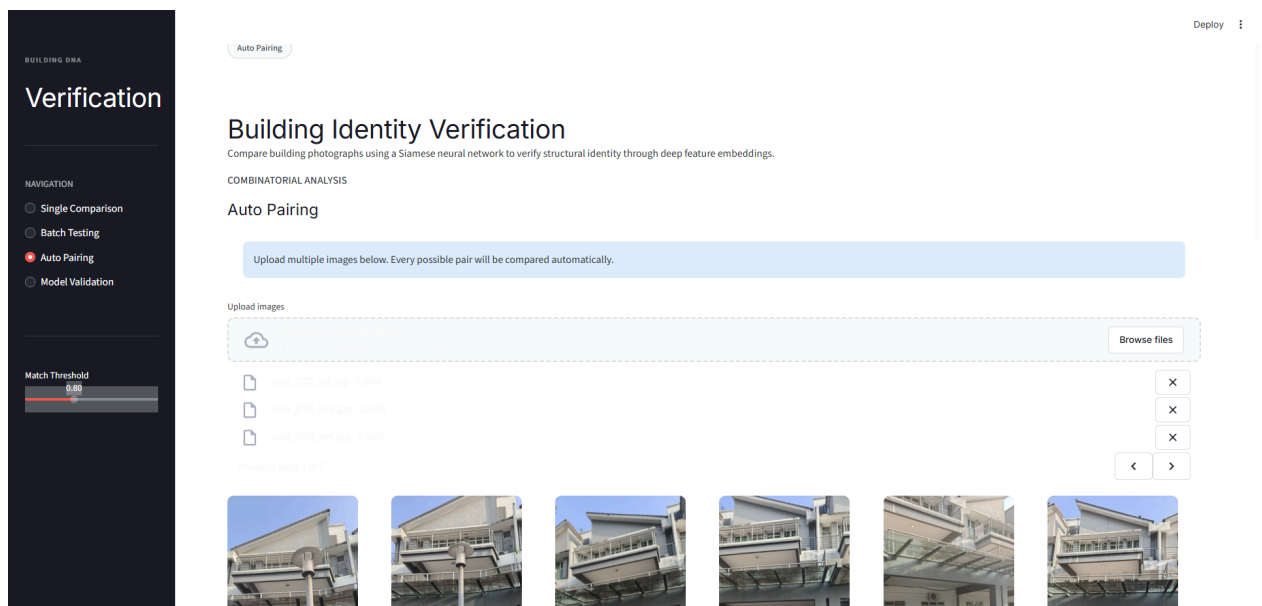


Figure 5.4.15 Showcase the in-app screenshot of the Batch Testing feature

3. Auto Pairing

A datasets of 20 images (10 pairs) that are evaluated with the top 5 highest and lowest scoring is constructed for this auto-pairing feature.



BUILDING DNA

Verification

NAVIGATION

Single Comparison

Batch Testing

Auto Pairing

Model Validation

Match Threshold

0.80

pair_001_live.jpg

pair_001_ref.jpg

pair_002_live.jpg

pair_002_ref.jpg

pair_003_live.jpg

pair_003_ref.jpg

pair_004_live.jpg

pair_004_ref.jpg

pair_005_live.jpg

pair_005_ref.jpg

pair_006_live.jpg

pair_006_ref.jpg

pair_007_live.jpg

pair_007_ref.jpg

pair_008_live.jpg

pair_008_ref.jpg

pair_009_live.jpg

pair_009_ref.jpg

pair_010_live.jpg

pair_010_ref.jpg

Deploy

BUILDING DNA

Verification

NAVIGATION

Single Comparison

Batch Testing

Auto Pairing

Model Validation

Match Threshold

0.80

pair_007_live.jpg

pair_007_ref.jpg

pair_008_live.jpg

pair_008_ref.jpg

pair_009_live.jpg

pair_009_ref.jpg

pair_010_live.jpg

pair_010_ref.jpg

20 images → 190 pairs to compare.

Embedding 19/20

Deploy

BUILDING DNA

Verification

NAVIGATION

Single Comparison

Batch Testing

Auto Pairing

Model Validation

Match Threshold

0.80

pair_001_live.jpg ↔ pair_009_live.jpg — REJECTED (Distance: 1.1191)

pair_001_live.jpg ↔ pair_009_ref.jpg — REJECTED (Distance: 1.4342)

pair_001_live.jpg ↔ pair_010_live.jpg — REJECTED (Distance: 1.7011)

pair_001_live.jpg ↔ pair_010_ref.jpg — MATCH (Distance: 0.4831)

pair_001_live.jpg

pair_010_ref.jpg

Euclidean Distance

0.4831

MATCH — Same Building

Distance 0.4831 is below threshold 0.80

pair_001_ref.jpg ↔ pair_002_live.jpg — MATCH (Distance: 0.4027)

Deploy

Bachelor of Computer Science (Honours)
Faculty of Information and Communication Technology (Kampar Campus), UTAR

128

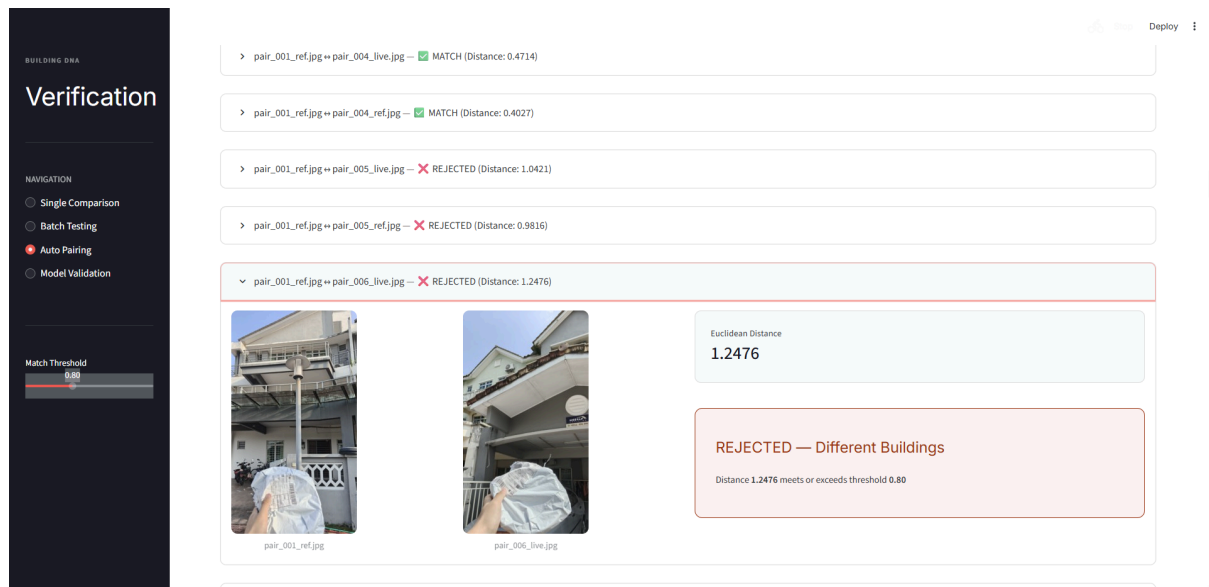


Figure 5.4.16, 5.4.17, 5.4.18, 5.4.19 Showcase the in-app screenshot of the Auto-Pairing feature

4. Model Validation

Performs the validation and generates reporting as describe in Chapter 4

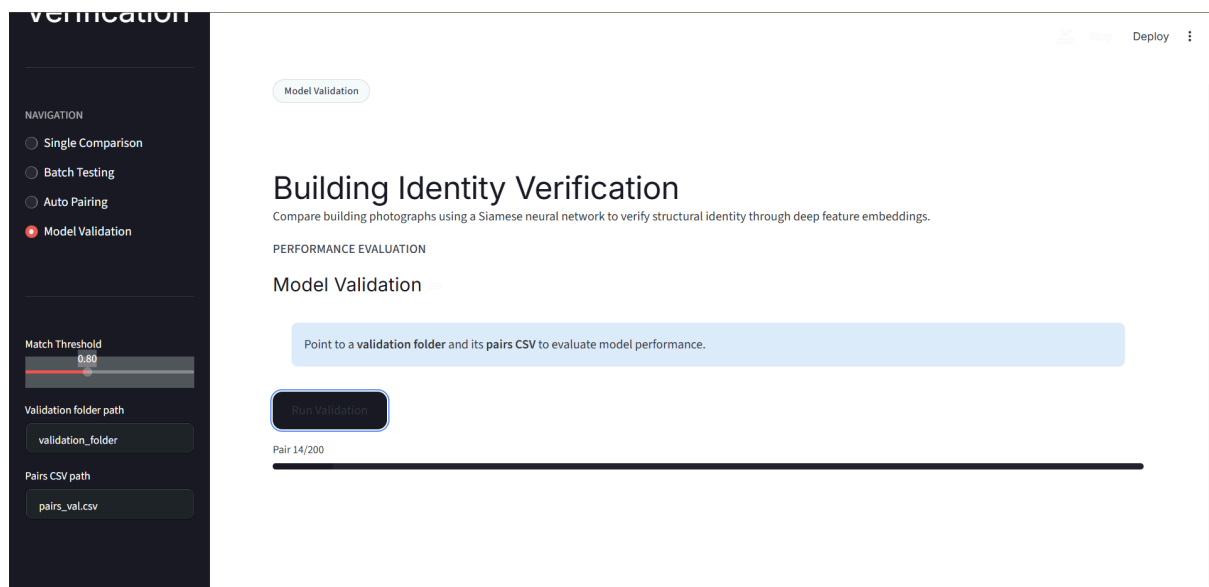


Figure 5.4.20 Showcase the in-app screenshot of images uploading to Model Validation tab

Evaluation and Testing of the Siamese Model

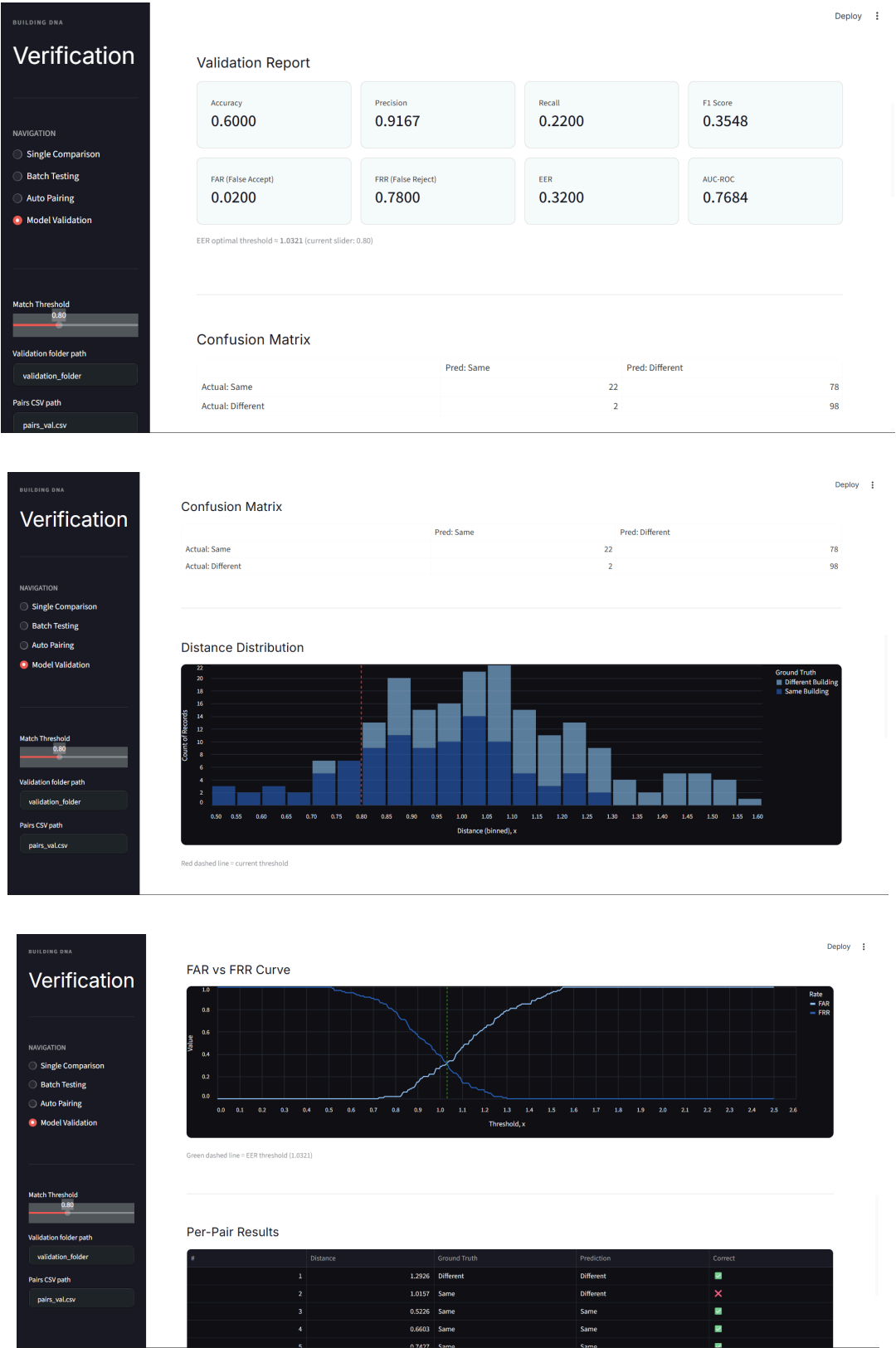


Figure 5.4.21, 5.4.22, 5.4.23 Showcase the in-app screenshot of the reporting and analysis done in Model Validation tab

5.5 Implementation Issues and Challenges

This section delineates the technical hurdles encountered during development, transitioning from a theoretical architecture to an operational edge-native mobile application required navigating unexpected build environment volatilities, extensive manual data curation, and complex machine learning discrepancies. The following subsections detail the primary hurdles encountered during the system's implementation.

5.5.1 Build Environment Volatility and Configuration Management

A persistent technical challenge encountered during the mobile application development phase was the instability of the native Android build environment. Specifically, the application encountered recurrent prebuild errors associated with the `local.properties` configuration file.

Within the React Native ecosystem, the `local.properties` file plays a crucial role in defining local environment variables, including the path to the Android SDK essential for compilation. However, during the automated prebuild process, a prerequisite for linking native hardware modules such as the Vision Camera and Fast-TFLite libraries, the execution scripts repeatedly overwrote or entirely deleted this file. This engendered a volatile Continuous Integration/Continuous Deployment (CI/CD) cycle, wherein subsequent builds failed due to missing SDK references. Mitigation necessitated extensive debugging of the build scripts to identify auto-generation conflicts and the enforcement of rigorous configuration management protocols to ensure the persistence of environment variables across native compilation cycles.

5.5.2 Logistical Bottlenecks in Proprietary Dataset Curation

Developing a specialized semantic building recognition model required a dataset reflecting the architectural homogeneity of Malaysian residential areas. With no suitable open-source datasets available, a proprietary dataset was curated from scratch.

Data acquisition was highly resource-intensive, involving in-situ photography of hundreds of unique structures in the Kampar district under varied lighting, angles, and distances, followed by sorting, cleaning, and manual annotation for the Neural Architecture Search pipeline. This

process consumed a large portion of the project timeline, underscoring the logistical challenges of custom computer vision in niche domains.

5.5.3 Domain Shift and the Semantic Gap in Operational Testing

The most critical machine learning challenge occurred during the evaluation of the Siamese Neural Network. Following the initial training phase, the model exhibited high accuracy on validation datasets but suffered a severe degradation in performance during simulated real-world operational testing. Analysis revealed this failure was caused by a phenomenon known as **Domain Shift**.

The initial dataset used clean, unobstructed photos of building exteriors and foyers. In real-world POD scenarios, images include "operational noise," such as the delivery parcel and driver's hand. As the Siamese Network was not trained on these occlusions during contrastive learning, it failed to generate accurate 128-dimensional structural embeddings, misclassifying identical buildings.

This domain shifting between the sterile training data and operational conditions halted deployment. Mitigation required revising data collection with deliberate "object injection" to retrain the network, enabling it to ignore noise and focus on permanent architectural features.

5.6 Concluding Remarks

The implementation phase of the "Standalone Package Delivery Verification" system represents a critical milestone in bridging the gap between theoretical architecture and a functional, real-world application. Throughout this chapter, the transformation of the project from high-level design into a cohesive ecosystem comprising mobile edge computing and cloud-based management has been detailed. By leveraging the Expo Application Service and React Native, the development successfully navigated the complexities of cross-platform deployment while maintaining the performance required for real-time computer vision tasks.

The integration of the various system operations, ranging from the dual-layer overlay and barcode OCR to the sophisticated Geolocation Gating Algorithm, demonstrates a multi-layered approach to delivery security. A significant achievement in this phase was the successful deployment of the TensorFlow Lite building detection model for edge-side

processing, ensuring that visual verification occurs instantaneously on the driver's device. Furthermore, the implementation of EXIF metadata injection logic and the Siamese Network for temporal image validation provides the system with a "data-centric" security layer, making the Proof of Delivery records both tamper-evident and geographically verifiable.

While the development journey faced significant implementation challenges, such as build environment volatility and the semantic gaps inherent in training models on proprietary logistical datasets, these hurdles were overcome through iterative refinement and robust configuration management. The resolution of these issues has resulted in a stable system ready for empirical scrutiny. With the implementation of the core capture modules, backend synchronization via Supabase, and the evaluation interface now complete, the project moves forward into Chapter 6. The subsequent phase will focus on rigorous lab evaluations and live field testing with GDEX to quantify the system's accuracy, latency, and overall efficacy in reducing delivery disputes.

CHAPTER 6 System Evaluation, Field Testing, and Discussion

This section describes the verification to test the integrity and accuracy of the metadata injected, as well as the overall system stability.

6.1 Verification of the EXIF Tags

The verification of EXIF tags is done via an external website onlineexifviewer.io to read on the image metadata saved from the proposed application.

The figure below illustrates that the uploaded photograph has been successfully embedded with the EXIF tags generated by the system. The areas of interest, highlighted in blue, correspond to the GPSLatitude and GPSLongitude tags embedded within the image's EXIF data. These GPS coordinates were subsequently entered into Google Maps to verify the precise location referenced, as depicted in the search results below, which exactly match the site where the delivery image was captured. This demonstrates the accuracy and reliability of the EXIF geolocation tags.

The screenshot displays the onlineexifviewer.io interface. On the left, there are three sections: 'Camera Make and Model' showing 'OPPO - OPPO Find X8', 'Camera Location Details' showing a map with a blue pin at coordinates 4.3298511, 101.13456090000001, and 'Image Preview' showing a small photo. On the right, the 'All Photo EXIF Data' section lists various metadata fields and their values.

All Photo EXIF Data	
ImageLength	4096
Make	OPPO
Orientation	Undefined
DateTime	2026:04:25 00:22:05
GPS Info IFD Pointer	1708
YResolution	72
XResolution	72
ImageWidth	3072
Model	OPPO Find X8
Software	MediaTek Camera Application
ImageDescription	
YCbCrPositioning	co-sited
Exif IFD Pointer	346
ResolutionUnit	inches
ExifVersion	0
ApertureValue	1.80
ExposureBiasValue	0
ExposureProgram	Normal program
ColorSpace	sRGB

Figure 6.1.1 Shows the interface of the external website that read image EXIF data

6.2 System Assessment

Based on the preliminary deployment testing on an actual Android device, the system are assessed as shown:

Performance: The Geolocation Gating algorithm operates smoothly with minimum overhead. Although the application requires 2-3 seconds to “warm start” the GPS, it is neglectable and does not hinder operation efficiency. The injection of EXIF metadata is instant, and local persistence storage of the image is smooth.

Usability: The Dual-Layer Overlay design is intuitive and visually appealing. It successfully provides a robust structured framework when capturing proof images. The visual feedback (Red/Green/Yellow) according to geolocation accuracy is intuitive, and the text hints overlay allow users to fully understand the functionalities without reading complex error messages.

Stability: The mobile application maintains a consistent frame rate (60fps) during the active camera preview, validating the decision to develop this system in React Native’s Native Modules environment.

6.3 Lab Evaluation of the Building Detection ML Model

Following the completion of the Neural Architecture Search (NAS) and fine-tuning pipeline, the model’s classification efficacy was empirically evaluated before deployment. The evaluation was conducted on a dataset of 578 total images, rigorously segmented into 463 training images, 58 validation images, and 57 test images.

6.3.1 Mathematical Evaluation Metrics

To quantify the model's reliability, performance was measured using two primary evaluation metrics: Precision and Recall. In the context of a Zero-Trust logistics system, these metrics dictate the balance between operational security (avoiding fake deliveries) and operational convenience (avoiding driver frustration).

Precision mathematically defines the proportion of positive identifications that were actually correct, minimizing False Positives:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

Recall mathematically defines the proportion of actual positive instances that the model correctly identified, minimizing False Negatives:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

6.3.2 Performance Results and Confusion Matrix Analysis

The fine-tuned architecture demonstrated exceptional predictive capability, achieving an Average Precision (PR AUC) of 0.998 and a minimal Log loss of 0.114. At the baseline confidence threshold of 0.5, the model established a highly balanced equilibrium, recording a Precision of 96.5% and a Recall of 96.5%.

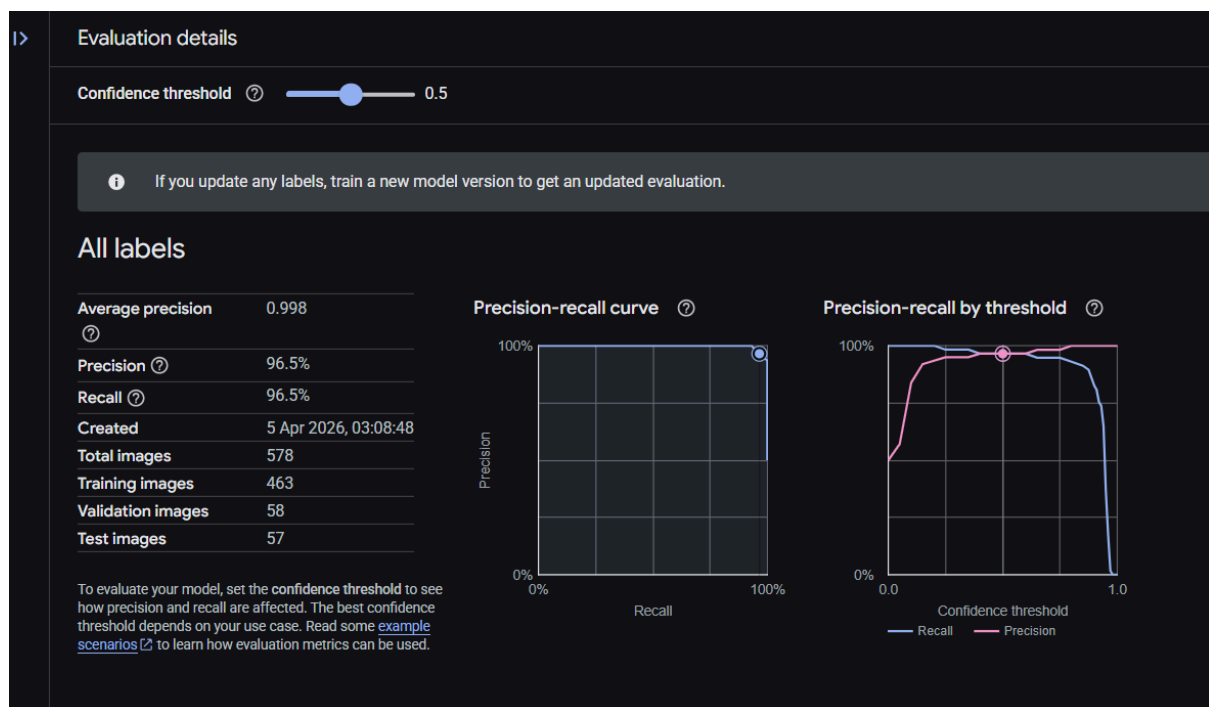


Figure 6.3.1 The evaluation details report shown in Google Vertex AI Platform

Further analysis of the Confusion Matrix confirms the model's robustness against the injected environmental noise. When presented with valid architectural structures (**building_detected**), the model successfully classified them with 98% accuracy, yielding a negligible 2% false negative rate. Conversely, when evaluating true negative environments (**no_building**), the model successfully rejected 93% of the instances, with only a 7% false positive rate. This proves the model successfully learned to isolate permanent architectural features rather than relying on the augmented delivery parcels.

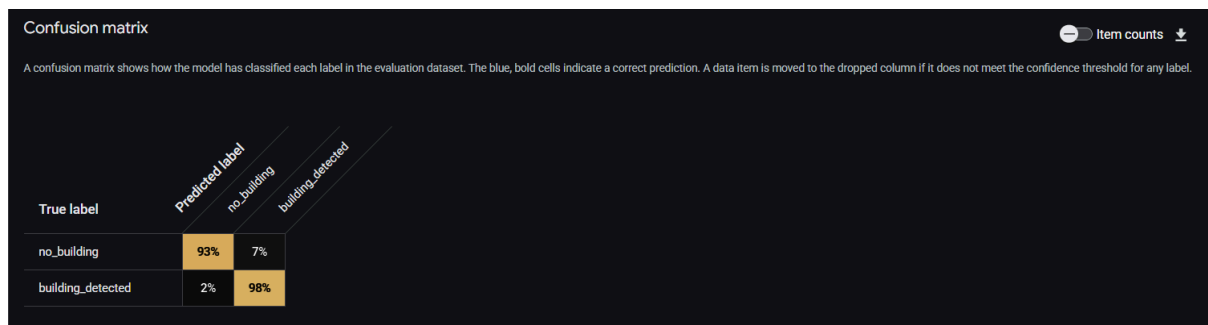


Figure 6.3.2 Confusion matrix of the Building Detection model

Prior to initiating the edge quantization pipeline, the fine-tuned architecture was temporarily deployed to a Google Vertex AI cloud endpoint. This facilitated a rigorous qualitative verification phase utilizing the platform's "Deploy and Test" interface. The objective of this phase was to manually stress-test the model's heuristic boundaries by ingesting novel, unseen visual payloads that deliberately mimic ambiguous operational edge cases.

6.3.3 Cloud Endpoint Deployment and Qualitative Edge-Case Verification

Prior to initiating the edge quantization pipeline, the fine-tuned architecture was temporarily deployed to a Google Vertex AI cloud endpoint. This facilitated a rigorous qualitative verification phase utilizing the platform's "Deploy and Test" interface. The objective of this phase was to manually stress-test the model's heuristic boundaries by ingesting novel, unseen visual payloads that deliberately mimic ambiguous operational edge cases.

To empirically validate the model's semantic understanding, four distinct spatial configurations were manually tested:

Baseline Exterior (True Positive): shown in Figure below

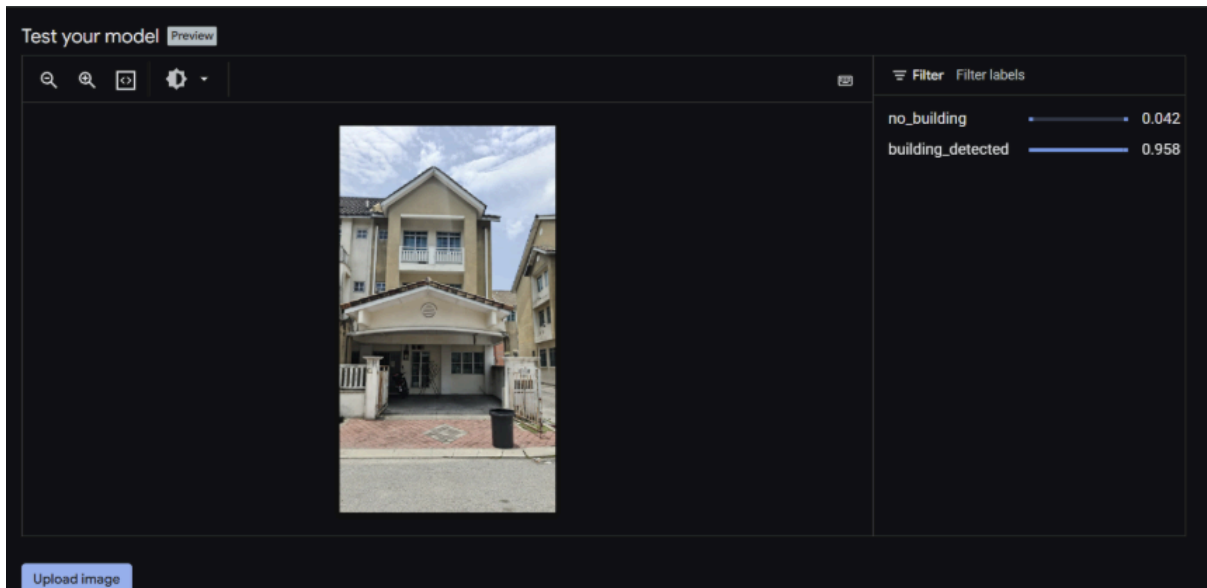


Figure 6.3.3 Front facing housing and performance of the model

An unobstructed image of a frontal housing facade. The model correctly computed a high-confidence (0.958) **building_detected** probability, verifying its baseline capacity to recognize macro-architectural topologies.

Micro-Environment (True Positive):

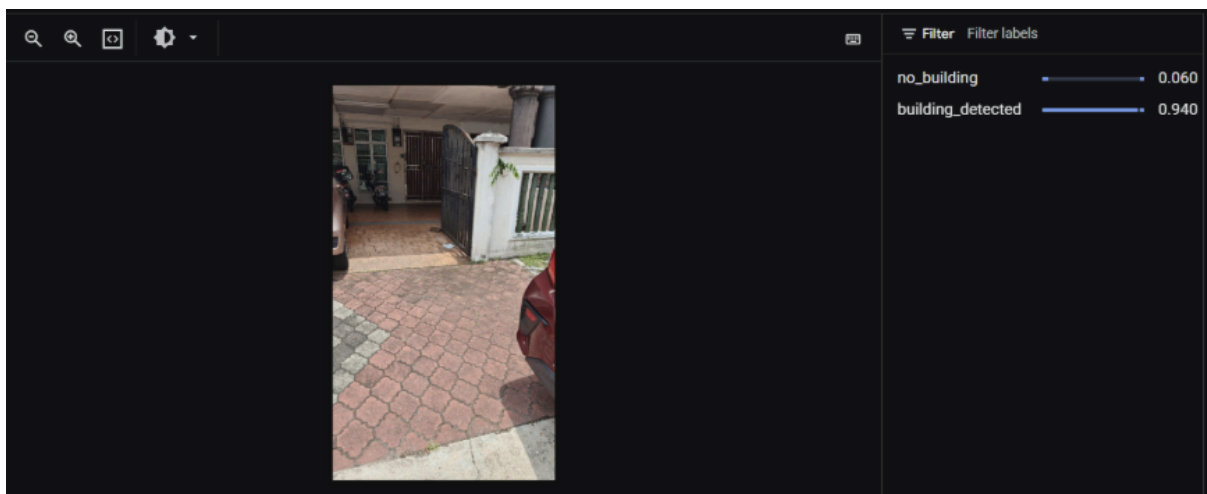


Figure 6.3.4 Performance of the model on half housing and half foyer

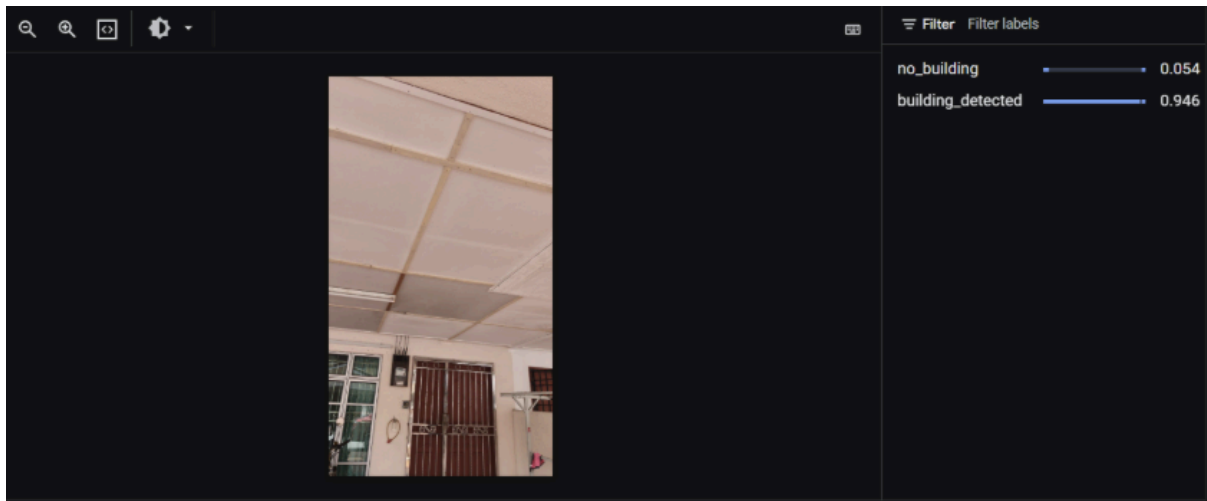


Figure 6.3.4 Performance of the model on building foyer

Image of a structurally confined interior foyer. The model successfully classified this as **building_detected**, proving it does not strictly rely on exterior perimeter gates to validate a delivery location.

Broad Environmental Vista (Clean Negative):

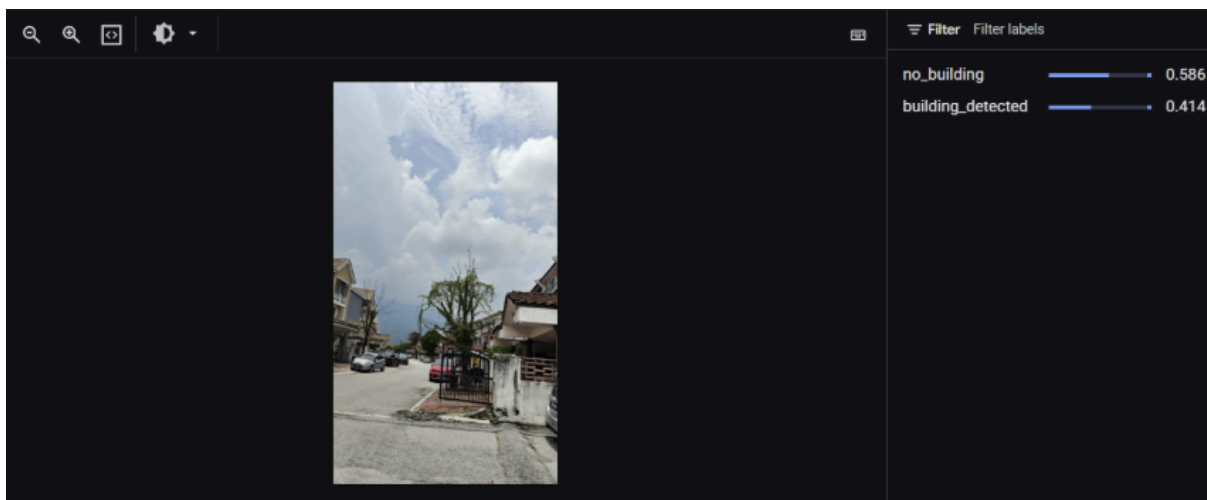


Figure 6.3.5 Performance of the model on a clear sky, road view Image

A wide road view dominated by sky, containing only distant, incomplete building geometries. The model successfully rejected the payload (**no_building**), demonstrating its ability to penalize fragmented background structures and prioritize immediate foreground architecture.

Artifact Isolation (Hard Negative):

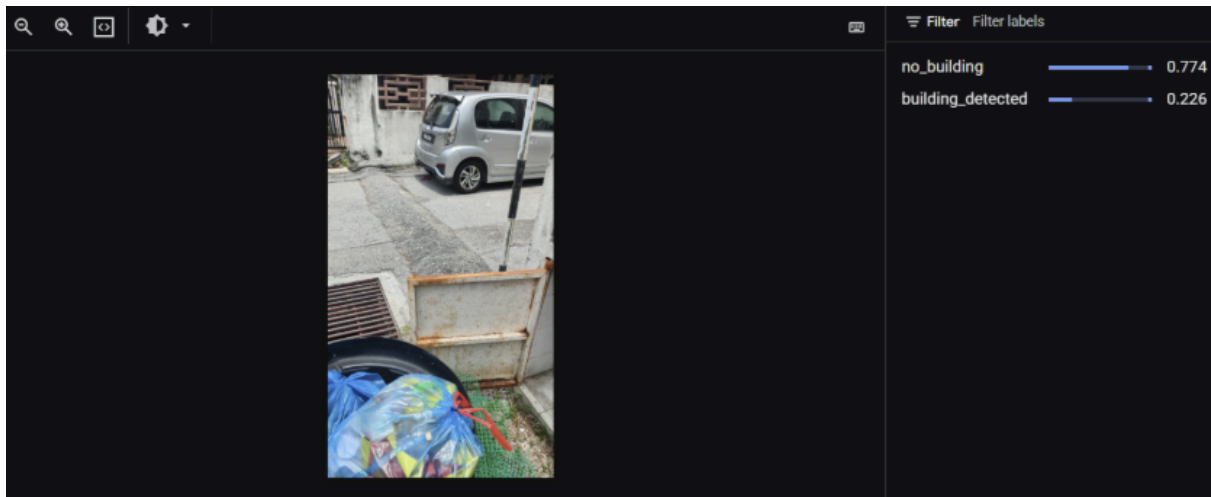


Figure 6.3.6 Performance of the model on non-building architecture

A close-up image of a standalone trash receptacle situated immediately outside a building. The model correctly classified this as **no_building**. This was a critical security validation, proving the network isolates complete architectural semantics rather than being mathematically triggered by isolated, generic man-made textures.

6.4 Evaluation and Testing of the Siamese Model

With referring to Figure 5.4.21, 5.4.22, 5.4.23, the Siamese network was evaluated against the curated validation dataset using the custom diagnostic dashboard. To establish a baseline security posture, the initial Euclidean distance threshold was strictly configured to 0.80. At this threshold, the model demonstrated exceptional defensive capabilities, yielding a Precision of 0.9167 and a False Accept Rate of precisely 0.0200. The confusion matrix confirms that out of 100 negative pairs, the model successfully rejected 98 pairs. This proves the architecture is highly resistant to fraudulent delivery spoofing.

However, this aggressive security configuration introduced a significant operational trade-off. The strict 0.80 threshold resulted in a high False Reject Rate of 0.7800 and a low Recall of 0.2200, successfully matching only 22 of the 100 positive pairs. As visualized in the distance distribution histogram, the natural variance of identical buildings frequently produces distance scores higher than 0.80. In a real-world deployment, this strict cut-off would cause high driver friction by rejecting valid Proof of Delivery captures.

To resolve this imbalance, the evaluation utilized the Equal Error Rate to determine the mathematically optimal decision boundary. The system recorded an EER of 0.3200 and an overall AUC-ROC of 0.7684, indicating acceptable model separability despite the extreme architectural homogeneity of the Kampar dataset. Crucially, the diagnostic system calculated the optimal operational threshold to be approximately 1.0321. Shifting the production deployment to this optimized threshold will significantly reduce the FRR, resolving the operational bottleneck while maintaining a scientifically defensible security boundary.

6.5 Live Operational Field Testing with GDEX

6.5.1 Objectives of Field Testing

The primary objective of this field test was to validate the "Delivery Pod" verification system's pipeline performance measures in a real-world scenario. While theoretical testing established the baseline functionality of the Machine Learning (ML) model and EXIF metadata extraction, physical validation was strictly required to test the algorithm against real-world anomalies. Specifically, the test aimed to evaluate the system's resilience to dynamic lighting conditions, GPS inaccuracies, and variable network latency during transit. Furthermore, the data collected served as a critical "Proof of Concept" (POC) dataset to tune the error threshold of the AI model.

6.5.2 Operational Scope & Logistics

The field test was conducted in formal collaboration with the GDEX Kampar Branch. The observational ride-along took place on April 25, 2026, operating from 09:00 to 13:00. The delivery route initiated in Kampar, Perak, and extended to Tapah, Perak, covering a mix of urban and semi-rural network coverage zones. Over the four-hour operational window, a total of **22 out of 27** parcels were successfully delivered and processed through the testing pipeline. The remaining five parcels failed delivery due to circumstances unrelated to the system, but attributable to standard GDEX operational procedures. The testing environment was maintained with 100% self-sufficiency; all necessary testing devices, power banks, and independent mobile network hotspots were independently provisioned to avoid reliance on GDEX hardware.

6.5.3 Execution Methodology & Protocols

Edge-Case Anomaly Mitigation Strategies

To accommodate unforeseen circumstances without hindering industry operations, a dedicated "Field Testing Mode" is activated by a continuous triple tap on the 'GPS Accuracy Label' in the Capture Screen. This mode bypasses all verification gates, including GPS Accuracy and Building Detection confidence scores. However, it logs additional data such as

upload latency, GPS time-to-fix, battery level, and more in the backend for later performance analysis (detailed database design described in Chapter 4).

- To prevent hallucinations in the Building Detection model, a “Ground Truth Option” is implemented (shown in the figure below), presenting a Yes/No option after each capture to confirm a building is present.
- To prevent unforeseen barcode undetectable issues, but in order to not hinder the operation, a sample consignment note is prepared, with a fixed parcel id: ‘CN25000907029’, the sample is shown in Figure 4.4.1

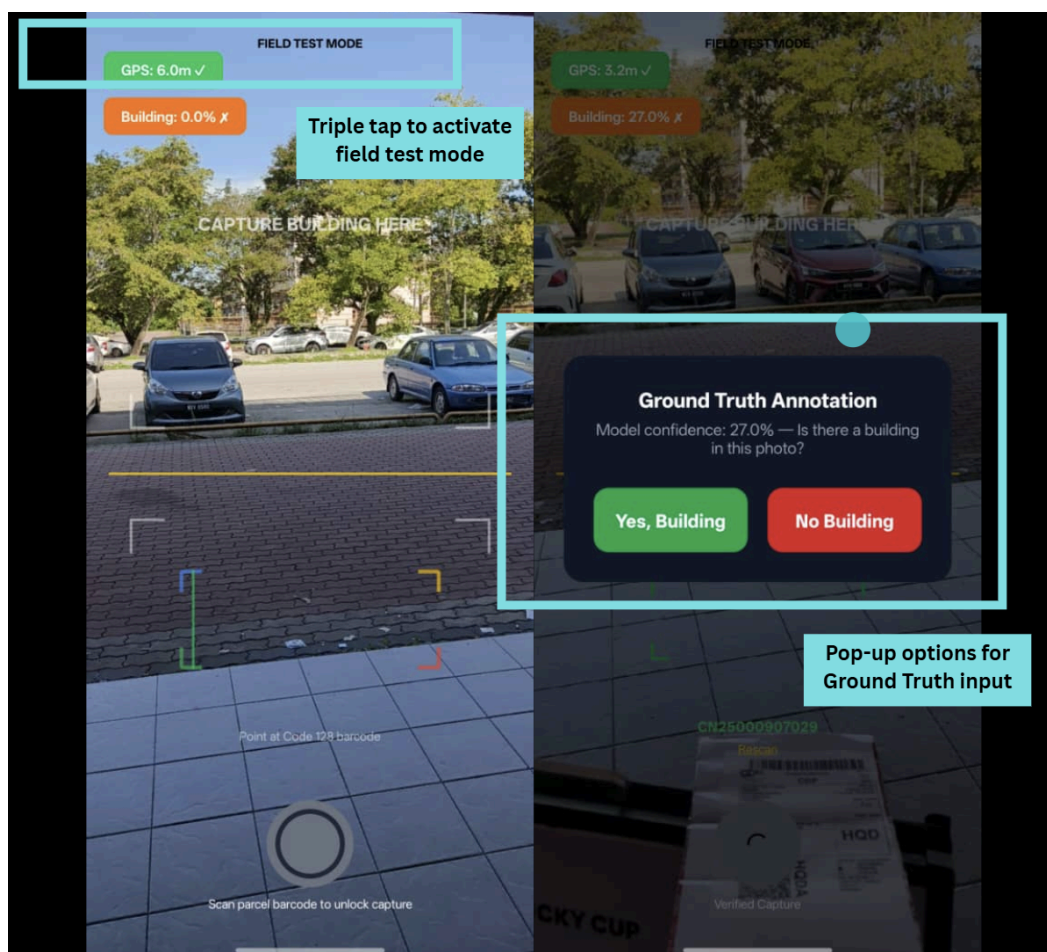


Figure 6.5.1 The ground truth options after capturing in Field Test mode

Safety Protocols and Conduct cater to GDEX SOP

To ensure zero disruption to operational efficiency, the testing was conducted under strict non-interference and safety protocols, with the researcher acting solely as a passive observer. At each of the 27 delivery nodes, the prototype executed a straightforward validation loop:

1. **Target Acquisition:** The device was pointed at the target building.
2. **Verification:** The AI evaluated and verified the building exterior.
3. **Data Capture:** An optical barcode scan was performed. To maintain strict privacy boundaries, the system was programmed to capture and process only alphanumeric data strings; no images of consignment notes, airway bills, or customer details were recorded.
4. **Synchronization:** The captured EXIF location data and alphanumeric strings were synced to the backend database.
5. **Iteration:** The system reset to proceed to the next delivery node.

6.5.4 System Performance

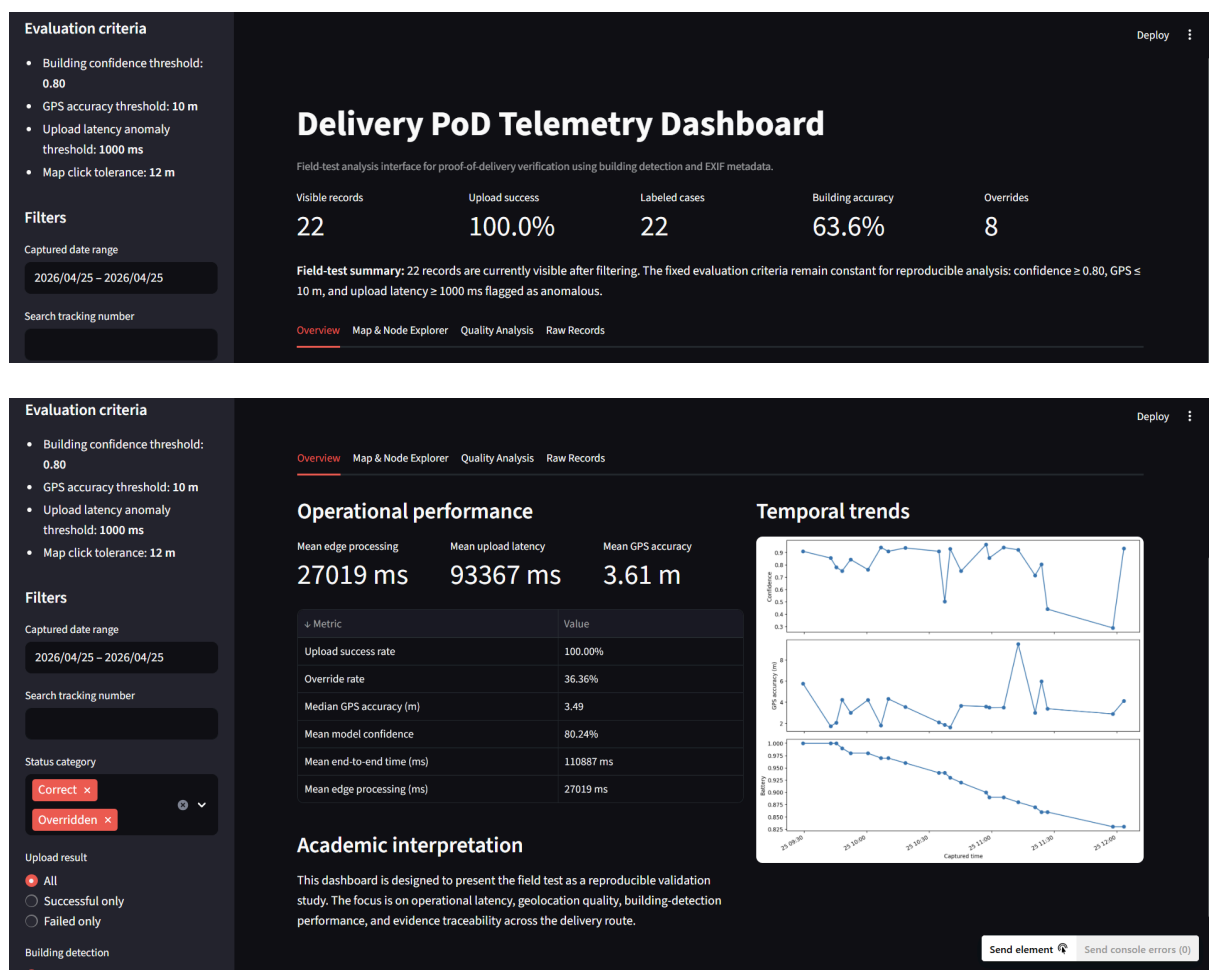


Figure 6.5.2, 6.5.3 showcase the report data captured on Field Test

The overall synchronization success rate was determined by the proportion of successfully uploaded records to the total number of captured delivery events. This metric assesses the

system's ability to reliably transmit delivery evidence under real-world mobile network conditions.

To evaluate operational responsiveness, the following aggregate metrics were examined:

- Mean upload latency
- Median upload latency
- 95th percentile (P95) upload latency
- Mean edge-processing duration
- GPS acquisition duration
- EXIF writing duration

The P95 upload latency metric was included to gauge worst-case performance under field conditions. Unlike mean latency, it captures high-latency outliers that could impact usability during real courier operations.

The edge-processing duration was calculated as:

where:

- is the barcode scanning duration,
- is the image capture duration,
- is the EXIF metadata writing duration.

System Performance Analysis

Analysis of 22 field-test records confirms system stability with full synchronization.

- **Upload Success Rate: 100%** (all records synchronized)
- **Human Overrides: 8 cases (36.36% rate)**, due to ambiguous conditions

Building Detection

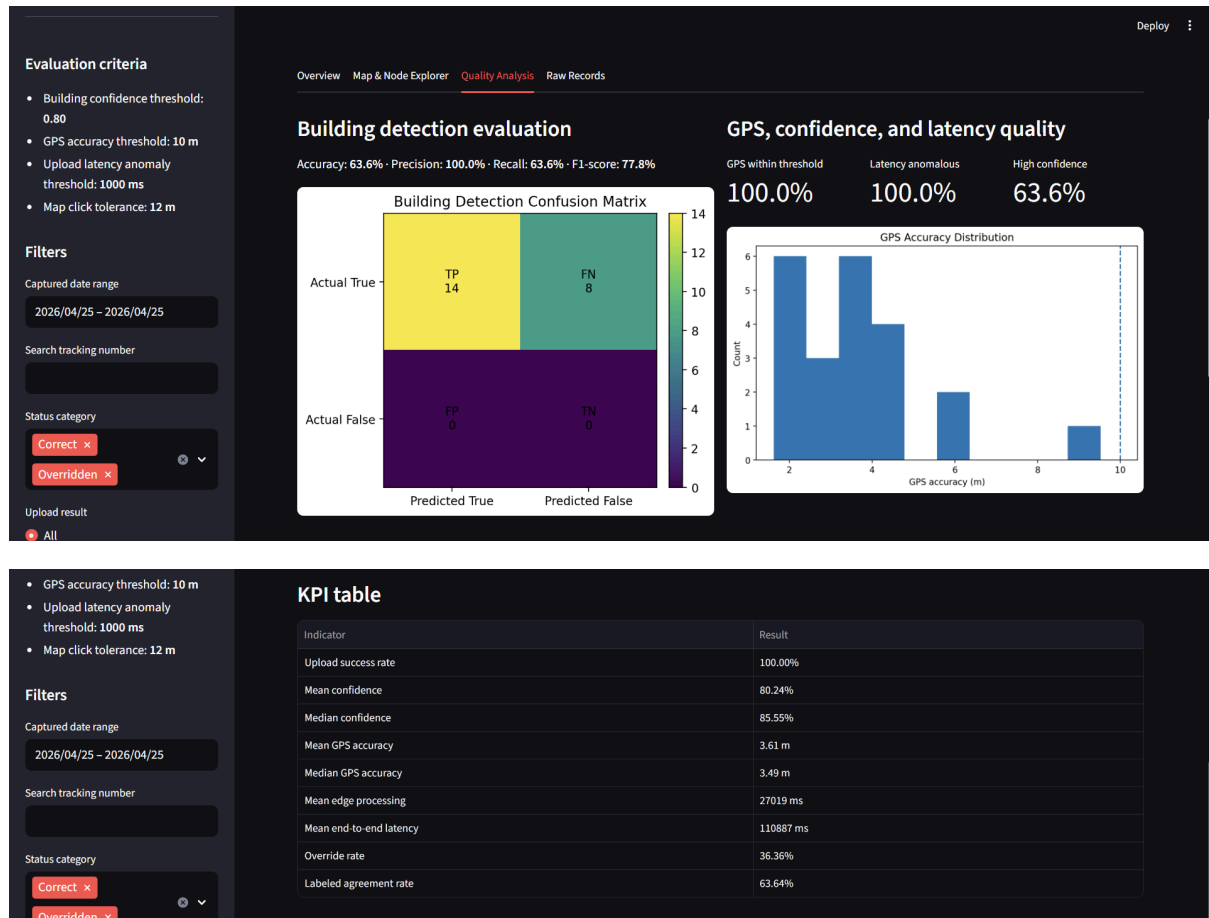


Figure 6.5.4, 6.5.5 showcase the Confusion Matrix and KPI table of the data captured in Field Test

The building-detection component was evaluated using the logged prediction outputs and manually verified ground-truth labels collected during the field test.

The following metrics were analyzed:

- Model confidence score
- Building detection result
- Ground-truth building label
- Human override occurrence

A confusion matrix was used to evaluate classification performance through:

- Accuracy: **63.64%**
- Precision: **100%** (0 false positives)

- Recall: **63.64%** (8 false negatives)
- F1-Score: **77.78%**
- Confusion Matrix: 14 TP, 8 FN, 0 FP, 0 TN

Confidence Scores

- Mean: **0.8024**
- Median: **0.8555**
- **63.64%** > 0.80 threshold

Processing Times (ms, Mean)

Several operational observations were identified during the experiment:

- GPS accuracy fluctuated depending on environmental conditions and vehicle movement.
- Upload latency varied according to network availability during transit.
- Low-confidence detections occasionally required human override validation.
- The metadata logging pipeline remained consistent throughout the field test.

GPS Accuracy (m)

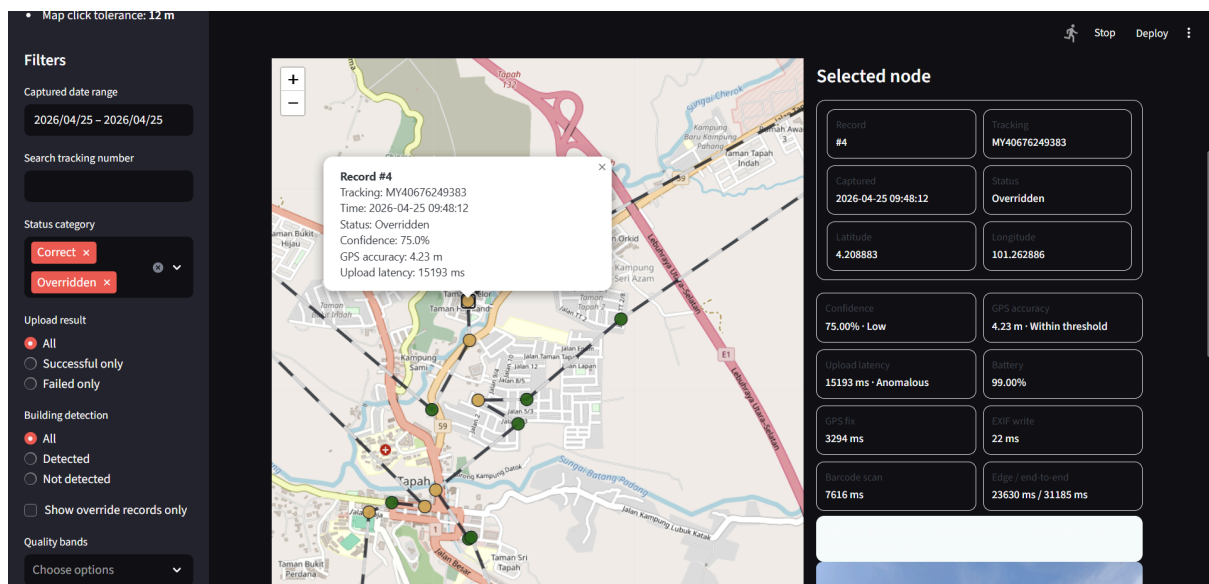


Figure 6.5.6 showcase the interactive map tailored for analysis of the Field Test route

GPS reliability was assessed using the recorded `gps_accuracy_metres` values. Lower GPS accuracy values indicate stronger positional precision and greater confidence in delivery location verification:

- Mean: **3.61**
- Median: **3.49**
- **100%** within 10m threshold

Overall: Reliable sync & GPS; moderate building recall; optimize upload latency.

6.5.5 Limitations and Future Improvements

Despite the successful results above, several edge cases and anomalies were identified that warrant improvements in future iterations:

- The barcode scanner fails if the barcode is upside down.
- Building confidence scores drop low when people stand in front of the camera or house.
- Consignment notes with imperfect barcodes are often undetectable; closer shots resolve this but lack the snappiness of scanning perfect barcodes.
- The system currently processes only single parcels; support for multiple consecutive scans in the same batch is unavailable.

CHAPTER 7 Conclusion and Recommendation

7.1 Conclusion

The last-mile delivery sector currently faces a significant "trust gap" due to the limitations of traditional Proof of Delivery systems. Conventional methods often rely on unstructured photographs and manual review of the POD images, which are susceptible to fraud and frequently fail to resolve customer disputes. This project successfully addressed these challenges by developing a standalone package delivery verification system that leverages forensic-style data extraction and advanced Computer Vision.

The project achieved its primary objectives through a multi-phase development approach:

- **System Architecture:** Using React Native and the Expo Application Service, a robust POD system was developed. The implementation of an Offline-First Architecture ensures that delivery personnel can capture evidence even in areas with poor connectivity, with data stored in local persistence before syncing to the Supabase cloud infrastructure.
- **Metadata Integration:** A core contribution of this work is the Geolocation Gating Algorithm and the Client-Side EXIF Metadata injection module. By embedding geolocation data and timestamps directly into the image file's metadata, the system creates a tamper-evident record that serves as a more robust legal proof than a standard image file.
- **Computer Vision & Deep Learning:** The system successfully integrated a Building Detection model optimized via Neural Architecture Search for edge deployment. Furthermore, the implementation of a Siamese Twin Architecture allows for temporal image validation, ensuring that the delivery environment matches historical records or pre-delivery staging.
- **Operational Validation:** Field testing conducted with GDEX demonstrated that the integration of a Dual-Layer Overlay and automated verification significantly reduces the ambiguity of delivery photos. The qualitative and quantitative results from

Chapter 6 confirm that these technical interventions effectively bridge the credibility gap between courier companies and consumers.

In summary, this research demonstrates that combining computer vision with deep metadata extraction provides a scalable, forensic approach to delivery verification that strengthens customer trust and operational accountability.

7.2 Research Limitations

While this project successfully demonstrates a multi-layered approach to delivery verification, there are specific technical and operational boundaries that define the current reach of the research. These limitations provide the context for the system's current performance and its future transition to an industrial tool.

7.2.1 Proof of Concept Status of the Siamese Network

A primary limitation of the current research is the deployment status of the Deep Learning Cross-Temporal Location Verification module. Although the Siamese Twin Architecture is fully functional, having been trained on a dataset of approximately 1,000 pairings from 100 distinct terrace housing units, it currently operates as a **Proof of Concept**. This module is implemented as a **functional independent Web UI** rather than being integrated directly into the mobile application's edge layer. Consequently, the high-dimensional feature distance calculations required to verify current POD images against historical baselines are conducted in a controlled environment to demonstrate technical feasibility.

7.2.2 Dependency on Historical Data and API Integration

The "Grounding Infrastructure" developed in this project links EXIF metadata as a foreign key to query historical records; however, the system does not yet feature **Real-Time Live Database Integration**. Full operational autonomy requires seamless access to GDEX's proprietary historical database, which encompasses millions of records across diverse geographical regions. Because this level of API access was out of scope for the current research phase, the system cannot yet autonomously trigger a "verified" or "rejected" status during an active delivery without human oversight.

7.2.3 Geographic and Environmental Constraints

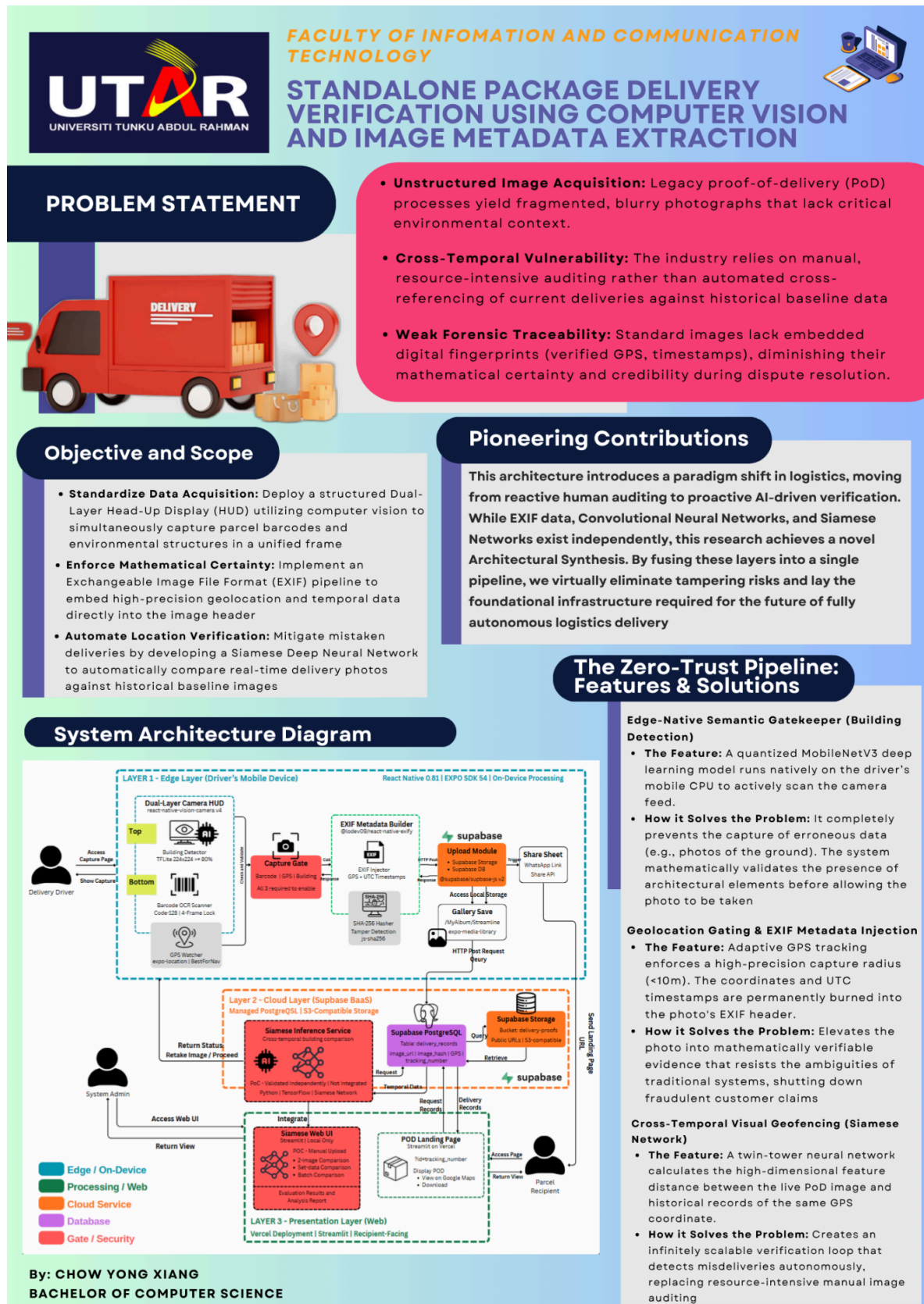
The current building detection and Siamese models are optimized specifically for residential environments, specifically terrace housing within Malaysian residential areas. This focus allowed for high accuracy within the defined scope but means the system is not yet prepared for a nationwide industrial rollout that would include high-rise apartments, industrial zones, or rural areas. The "Semantic Gap" and domain shift inherent in these varying environments remain a challenge that requires more diverse datasets for future mitigation.

7.2.4 Path to Full Autonomy

Currently, the system serves as an evidentiary tool rather than a fully autonomous decision-maker. While the Siamese Network can confirm if a delivery location matches a historical baseline to prevent misdeliveries [\[11\]](#), the "Fully Autonomous" trigger, where the AI independently authorizes a delivery drop is treated as a demonstration of potential. However, the architecture is designed to be highly scalable. Further collaboration with GDEX to integrate their proprietary historical data will allow this proof of concept to be transitioned into a live, mobile-integrated action in a very short timeframe.

APPENDIX

Poster



Interfaces

EXIF Metadata TypeScript Interfaces - exif.types.ts

```
/**
 * EXIF metadata TypeScript interfaces
 */
export interface EXIFMetadata {
  // Date/Time
  DateTimeOriginal?: string;
  DateTimeDigitized?: string;
  DateTime?: string;

  // GPS
  GPSLatitude?: number;
  GPSTimeStamp?: number;
  GPSAltitude?: number;
  GPSAltitudeRef?: number;
  GPSHeading?: number;
  GPSSpeed?: number;
  GPSTimeStamp?: string;
  GPSTimeStamp?: string;

  // Camera settings
  Make?: string;
  Model?: string;
  Orientation?: number;
  XResolution?: number;
  YResolution?: number;
  ResolutionUnit?: number;

  // Image properties
  ImageWidth?: number;
  ImageHeight?: number;
  ColorSpace?: number;
  PixelXDimension?: number;
  PixelYDimension?: number;

  // Exposure
  ExposureTime?: number;
  FNumber?: number;
```



```

    ExposureProgram?: number;
    ISOSpeedRatings?: number;
    ExposureBiasValue?: number;
    MeteringMode?: number;
    Flash?: number;

    // Focal length
    FocalLength?: number;
    FocalLengthIn35mmFilm?: number;

    // White balance
    WhiteBalance?: number;

    // Other
    Software?: string;
    Artist?: string;
    Copyright?: string;
}

export interface EXIFProcessingResult {
    success: boolean;
    metadata: EXIFMetadata | null;
    error?: string;
}

```

Location Related Interface - location.types.ts

```

/**
 * Location-related TypeScript interfaces for GPS coordinate
 * handling
 */

export interface LocationCoordinate {
    latitude: number;
    longitude: number;
    accuracy: number | null;
    altitude: number | null;
    altitudeAccuracy: number | null;
    heading: number | null;
    speed: number | null;
}

```

```

    timestamp: number;
  }

  export interface LocationWatcherOptions {
    accuracy?: 'lowest' | 'low' | 'balanced' | 'high' | 'highest'
    | 'bestForNavigation';
    timeInterval?: number;
    distanceInterval?: number;
  }

  export interface LocationWatcherState {
    coordinate: LocationCoordinate | null;
    isloading: boolean;
    error: Error | null;
    isWatching: boolean;
  }

  export interface LocationPermissionStatus {
    granted: boolean;
    canAskAgain: boolean;
    status: 'granted' | 'denied' | 'undetermined';
  }

```

/Utils

```

// api.ts
import * as FileSystem from 'expo-file-system/legacy';
import { EXIFMetadata } from '@types/exif.types';
import { supabase } from '@utils/supabase_client';
import { hashImageFile } from '@utils/hasher';

export interface CapturePayload {
  photoUri: string;
  location: {
    latitude: number;
    longitude: number;
    accuracy: number | null;
    altitude: number | null;
    timestamp: number;
  };
};

```

```

    exif: EXIFMetadata | null;
    accuracyStatus: 'precise' | 'override';
    trackingNumber: string;
}

/**
 * Uploads the proof image to Supabase Storage and inserts a
 * delivery record.
 * Also computes a SHA-256 hash of the image for tamper detection.
 */
export async function sendToAPI(payload: CapturePayload):
Promise<void> {
    try {
        // 1) Compute image hash for tamper detection
        const imageHash = await hashImageFile(payload.photoUri);

        // 2) Read image file as base64
        const fileBase64 = await
FileSystem.readAsStringAsync(payload.photoUri, {
    encoding: FileSystem.EncodingType.Base64,
});

        // 3) Upload image to Supabase Storage
        const fileName = `delivery_${Date.now()}.jpg`;
        const { data: storageData, error: storageError } = await
supabase.storage
    .from('delivery-proofs')
    .upload(fileName, decode(fileBase64), {
        contentType: 'image/jpeg',
        upsert: false,
    });

        if (storageError) throw new Error(`Storage upload failed:
${storageError.message}`);

        // 4) Get public URL of uploaded image
        const { data: urlData } = supabase.storage
    .from('delivery-proofs')
    .getPublicUrl(storageData.path);

        const imageUrl = urlData.publicUrl;

```

```

        // 5) Insert delivery record into database
        const { error: dbError } = await
supabase.from('delivery_records').insert({
    image_url: imageUrl,
    image_hash: imageHash,
    latitude: payload.location.latitude,
    longitude: payload.location.longitude,
    altitude: payload.location.altitude,
    gps_accuracy_metres: payload.location.accuracy,
    captured_at: new
Date(payload.location.timestamp).toISOString(),
    accuracy_status: payload.accuracyStatus,
    tracking_number: payload.trackingNumber,
});

    if (dbError) throw new Error(`Database insert failed:
${dbError.message}`);

    console.log('✅ Delivery record saved:', imageUrl);
} catch (error) {
    console.error('❌ sendToAPI failed:', error);
    throw error;
}
}

/**
 * Decodes a base64 string into a Uint8Array for Supabase Storage
upload.
 */
function decode(base64: string): Uint8Array {
    const binaryString = atob(base64);
    const bytes = new Uint8Array(binaryString.length);
    for (let i = 0; i < binaryString.length; i++) {
        bytes[i] = binaryString.charCodeAt(i);
    }
    return bytes;
}

```

References

- [1]
S. Arora, V. Choudhary, and P. Kireyev, “Don’t Fake It If You Can’t Make It: Driver Misconduct in Last-Mile Delivery,” *Management Science*, Aug. 2024, doi: 10.1287/mnsc.2023.01829.
- [2]
L. B. Elvas, U. Tokkozhina, A. Martins, and J. C. Ferreira, “Implementation of disruptive technologies for the last mile delivery efficiency achievement,” *Transportation research procedia*, vol. 72, pp. 32–39, Jan. 2023, doi: 10.1016/j.trpro.2023.11.319.
- [3]
H. S. An, A. Park, J. M. Song, and C. Chung, “Consumers’ adoption of parcel locker service: protection and technology perspectives,” *Cogent Business & Management*, vol. 9, no. 1, Nov. 2022, doi: 10.1080/23311975.2022.2144096.
- [4]
N. Soni, “Forensic Value of Exif Data: An Analytical Evaluation of Metadata Integrity across Image Transfer Methods,” *Perspectives in Legal and Forensic Sciences*, vol. 2, no. 2, pp. 10006–10006, Jan. 2025, doi: 10.70322/plfs.2025.10006.
- [5]
J. Du, W. Fu, Y. Zhang, and Z. Wang, “Advancements in Image Recognition: A Siamese Network Approach,” *Information Dynamics and Applications*, vol. 3, no. 2, pp. 89–103, Jun. 2024, doi: 10.56578/ida030202.
- [6]
P. L. Mazzeo, C. Libetta, P. Spagnolo, and C. Distanto, “A Siamese Neural Network for Non-Invasive Baggage Re-Identification,” *Journal of Imaging*, vol. 6, no. 11, pp. 126–126, Nov. 2020, doi: 10.3390/jimaging6110126.
- [7]
D. P. Gangwar and A. Pathania, “Authentication of Digital Image using Exif Metadata and Decoding Properties,” *International Journal of Scientific Research in Computer Science Engineering and Information Technology*, pp. 335–341, Dec. 2018, doi: 10.32628/cseit183815.
- [8]
K. Rahman, “A Narrative Literature Review and E-Commerce Website Research,” *ICST Transactions on Scalable Information Systems*, vol. 5, no. 17, pp. 154806–154806, May 2018, doi: 10.4108/eai.29-5-2018.154806.
- [9]
A. Dharaningtyas and A. Nizar, “An analysis on the improvement of system for proof of delivery,” *International Journal of Research in Business and Social Science (2147-4478)*, vol. 10, no. 3, pp. 53–64, May 2021, doi: 10.20525/ijrbs.v10i3.1091.
- [10]

D. Prithvirajan and M. Mathirajan, “LPG Cylinder Delivery Authentication using Artificial Intelligence in Last Mile Delivery,” Sep. 2023, doi: 10.46254/ap04.20230129.

[11]

S. Hu and G. H. Lee, “Image-Based Geo-Localization Using Satellite Imagery,” *International Journal of Computer Vision*, vol. 128, no. 5, pp. 1205–1219, Jun. 2019, doi: 10.1007/s11263-019-01186-0.

[12]

R. T. Jagarlamudi, “Leveraging Computer Vision and IoT for Optimized Last-Mile Delivery in Smart Logistics Systems,” pp. 147–152, Dec. 2025, doi: 10.1109/icicnis66685.2025.11315516.

[13]

A. Naumann, F. Hertlein, L. Dörr, and K. Furmans, “TAMPAR: Visual Tampering Detection for Parcel Logistics in Postal Supply Chains,” vol. 2017, pp. 8061–8071, Jan. 2024, doi: 10.1109/wacv57701.2024.00789.

[14]

A. Naumann, F. Hertlein, L. Dörr, and K. Furmans, “TAMPAR: Visual Tampering Detection for Parcels Logistics in Postal Supply Chains,” *arXiv (Cornell University)*, Nov. 2023, doi: 10.5281/zenodo.10057089.

[15]

A. Naumann, F. Hertlein, L. Dörr, and K. Furmans, “TAMPAR: Visual Tampering Detection for Parcel Logistics in Postal Supply Chains,” *ArXiv.org*, Oct. 2023, doi: 10.48550/arxiv.2311.03124.

[16]

H. Ulrich *et al.*, “Understanding the Nature of Metadata: Systematic Review,” *Journal of Medical Internet Research*, vol. 24, no. 1, Oct. 2021, doi: 10.2196/25440.

[17]

N. A. A. Et.al, “Laboratory Instruments’ Produced Scientific Data Standardization through the Use of Metadata,” *Türk bilgisayar ve matematik eğitimi dergisi*, vol. 12, no. 3, pp. 2146–2151, Apr. 2021, doi: 10.17762/turcomat.v12i3.1157.

[18]

F. Ciotti and F. Tomasi, “Formal Ontologies, Linked Data, and TEI Semantics,” *Journal of the Text Encoding Initiative*, Sep. 2016, doi: 10.4000/jtei.1480.

[19]

by P. M.-R. and H. S. Rzepa, “Markup Languages—How to Structure Chemistry-Related Documents,” *Chemistry International*, vol. 24, no. 4, Jan. 2002, doi: 10.1515/ci.2002.24.4.9.

[20]

J. Zhang and S. M. Bellovin, “Preventing Intimate Image Abuse Via Privacy-Preserving Credentials,” *SMU Science and Technology Law Review*, vol. 26, no. 2, pp. 149–149, Jan. 2023, doi: 10.25172/smustr.26.2.2.

- [21]
G. Blewitt, “GPS, Reference Systems,” 2016, pp. 1–4. doi: 10.1007/978-3-319-02370-0_92-1.
- [22]
E. M. Japara, S. Arifin, and E. Irwansyah, “Geolocation System Module Creation to Validate User Location Coordinates in an Android Application Using Flutter Framework,” *Journal of Computer Science*, vol. 19, no. 8, pp. 1050–1064, Aug. 2023, doi: 10.3844/jcssp.2023.1050.1064.
- [23]
M. O. Thomas and B. B. Rad, “Reliability Evaluation Metrics for Internet of Things, Car Tracking System: A Review,” *International Journal of Information Technology and Computer Science*, vol. 9, no. 2, pp. 1–10, Jan. 25, 2017. doi: 10.5815/ijitcs.2017.02.01.
- [24]
E. G. Anderson, S. Y. Lim, and N. Joglekar, “Are More Frequent Releases Always Better? Dynamics of Pivoting, Scaling, and the Minimum Viable Product,” in *Proceedings of the ... Annual Hawaii International Conference on System Sciences/Proceedings of the Annual Hawaii International Conference on System Sciences*, Jan. 2017. doi: 10.24251/hicss.2017.705.
- [25]
H. Sharp and T. Hall, *Agile Processes, in Software Engineering, and Extreme Programming*. Springer Science+Business Media, 2016. doi: 10.1007/978-3-319-33515-5.
- [26]
A. Hussain, B. Barua, A. Osman, R. Abozariba, and A. T. Asyhari, “Performance of MobileNetV3 Transfer Learning on Handheld Device-based Real-Time Tree Species Identification,” in *2022 27th International Conference on Automation and Computing (ICAC)*, Sep. 2021, pp. 1–6. doi: 10.23919/icac50006.2021.9594222.
- [27]
M. Dhar, “Object Detection using Deep Learning Approach,” *International Journal for Research in Applied Science and Engineering Technology*, vol. 10, no. 6, pp. 2963–2969, Jun. 2022, doi: 10.22214/ijraset.2022.44417.
- [28]
S. Duffner, “Weakly Supervised and On-line Machine Learning for Object Tracking and Recognition in Images and Videos,” *HAL (Le Centre pour la Communication Scientifique Directe)*, Apr. 2019, Accessed: Oct. 2025. [Online]. Available: <https://theses.hal.science/tel-02157568>
- [29]
J. J. L. D. Olmo, P. E. López-de-Teruel, A. Ruiz-Jimeno, and F. J. Garcia-Clemente, “Computer Vision on the Edge: A Scalable Auto-ID Solution for Industrial Logistics,” *Procedia Computer Science*, vol. 265, pp. 276–284, Jan. 2025, doi: 10.1016/j.procs.2025.07.182.
- [30]

K. W. Goh, B. U. I. Khan, A. R. Khan, D. S. Putra, S. Sankaranarayanan, and Md. A. Bhuyian, “3L-BC: a three-layer blockchain architecture for collaborative machine learning in secure drone communications,” *Journal of King Saud University - Computer and Information Sciences*, vol. 37, no. 8, Sep. 2025, doi: 10.1007/s44443-025-00197-x.

[31]

S. Yao, B. R. Ardabili, A. D. Pazho, G. A. Noghre, C. Neff, and H. Tabkhi, “From Lab to Field: Real-World Evaluation of an AI-Driven Smart Video Solution to Enhance Community Safety,” *arXiv (Cornell University)*, Dec. 2023, doi: 10.48550/arxiv.2312.02078.

[32]

A. Hannan *et al.*, “A decentralized hybrid computing consumer authentication framework for a reliable drone delivery as a service,” *PLoS ONE*, vol. 16, no. 4, Apr. 2021, doi: 10.1371/journal.pone.0250737.

[33]

J. J. Losada-del-Olmo, E. Ballesteros, Á. L. P. Gómez, P. E. López-de-Teruel, A. Ruiz, and F. J. G. Clemente, “Fog computing-driven logistics: leveraging few-shot learning and foundational computer vision models,” *Cluster Computing*, vol. 28, no. 14, Oct. 2025, doi: 10.1007/s10586-025-05662-w.

[34]

R. Pierdicca *et al.*, “Smart maintenance of riverbanks using a standard data layer and Augmented Reality,” *Computers & Geosciences*, vol. 95, pp. 67–74, Jul. 2016, doi: 10.1016/j.cageo.2016.06.018.

[35]

M. Shastri and U. Shrivastav, “Optimizing Delivery Logistics: Enhancing Speed and Safety with Drone Technology,” *arXiv (Cornell University)*, Jul. 2025, doi: 10.48550/arxiv.2507.17253.