# IoT-Based Intelligent Harvest Sorting and Quality Analysis System

A Microservices Architecture Integrating Classical Computer Vision, Deep Learning, and Generative AI for Automated Crop Quality Control

**Author Name**
Department of Software Engineering
University Name
author@university.edu

Academic Year 2025 – 2026

**Abstract**

*(To be written after experimentation.)*

# Contents

# List of Figures

# List of Tables

# List of Acronyms

| **Acronym** | **Full Term** |
|---|---|
| CNN | Convolutional Neural Network |
| API | Application Programming Interface |
| IoT | Internet of Things |
| IIoT | Industrial Internet of Things |
| HSV | Hue, Saturation, Value |
| LLM | Large Language Model |
| REST | Representational State Transfer |
| GAP | Global Average Pooling |

# Chapter 1

# General Introduction

## 1.1 Context and Motivation

The date palm (*Phoenix dactylifera* L.) is one of the oldest cultivated fruit trees, with significant economic and cultural importance across the Middle East and North Africa.

According to the Food and Agriculture Organization of the United Nations (FAO), global date production reached approximately **10.09 million tonnes** in **2024**, with **Saudi Arabia**, **Egypt**, and **Algeria** being the leading producers [1].

Despite their economic value, dates are highly susceptible to quality degradation caused by fungal infection, insect infestation, excessive moisture, and mechanical damage during harvesting and transportation [2]. Post-harvest losses in the date sector are estimated at **30%** of the total annual yield in several producing regions [3].

In traditional packing facilities, quality grading is still predominantly performed by human inspectors who visually assess each fruit for colour, texture, size, and the presence of defects. This manual process is inherently slow, subjective, and non-reproducible: two different operators may assign different grades to the same fruit, and fatigue significantly degrades accuracy over long shifts [2].

The convergence of the Internet of Things (IoT), computer vision, and deep learning offers a promising path toward automating this process. Modern lightweight Convolutional Neural Networks (CNNs) such as MobileNetV2 can classify images with high accuracy while remaining deployable on resource-constrained environments [4]. When combined with classical image processing for object isolation and cloud-based inference services, a complete, non-destructive, and scalable quality control pipeline becomes feasible.

This project is therefore motivated by the need for an **end-to-end intelligent system** that can:

1. detect and isolate individual date fruits from a conveyor-belt camera feed,
2. classify each fruit's quality in real time, and

3. log, visualise, and *interpret* the production data through generative AI.

## 1.2 Problem Statement

Although several studies have explored CNN-based classification of date fruits [2], [5], most focus exclusively on the classification task in isolation. The broader engineering challenge of integrating detection, classification, data persistence, and operational decision support into a single deployable system remains largely unaddressed. The following specific gaps motivate the present work:

1. **Subjectivity of manual grading.** As discussed in Section 1.1, human inspectors produce inconsistent grades due to lighting, fatigue, and individual perception.

2. **Computational cost of detection-only deep learning pipelines.** End-to-end object-detection models such as YOLO [6] achieve high accuracy but require significant GPU resources. For scenarios where the background is controlled (e.g., a white conveyor belt), classical computer vision techniques can isolate objects at a fraction of the cost.

3. **Absence of integrated quality control workflows.** Existing research typically stops at reporting a classification accuracy score. A production-grade system must also handle image ingestion from an IoT device, persist predictions in a database, and surface actionable insights — not just raw numbers — to operations managers.

The central question addressed in this work can be formulated as:

> *How can a lightweight, cloud-deployable system be designed to automatically detect, classify, and log the quality of date fruits in real time, while providing AI-generated operational insights to production managers?*

## 1.3 Objectives

To answer the research question stated above, four operational objectives are defined:

**O1. Design a two-stage vision pipeline** that combines classical computer vision (colour-space transformation, Otsu's thresholding, morphological operations) for object isolation with a pre-trained MobileNetV2 CNN for quality classification.

**O2. Implement a containerised cloud API** using FastAPI and Docker, capable of receiving images over HTTP, performing inference entirely in memory, and returning predictions with sub-second latency.

**O3. Simulate an IoT edge device** that mimics a conveyor-mounted industrial camera, continuously capturing and transmitting images to the cloud service with built-in network resilience.

**O4. Persist inspection data and generate AI-driven quality reports** by logging every prediction to a PostgreSQL database (Supabase) and integrating a Large Language Model (Google Gemini) to interpret production trends and generate managerial recommendations.

## 1.4 Proposed Approach

The proposed system follows a four-layer microservices architecture, illustrated in Figure 1.1.

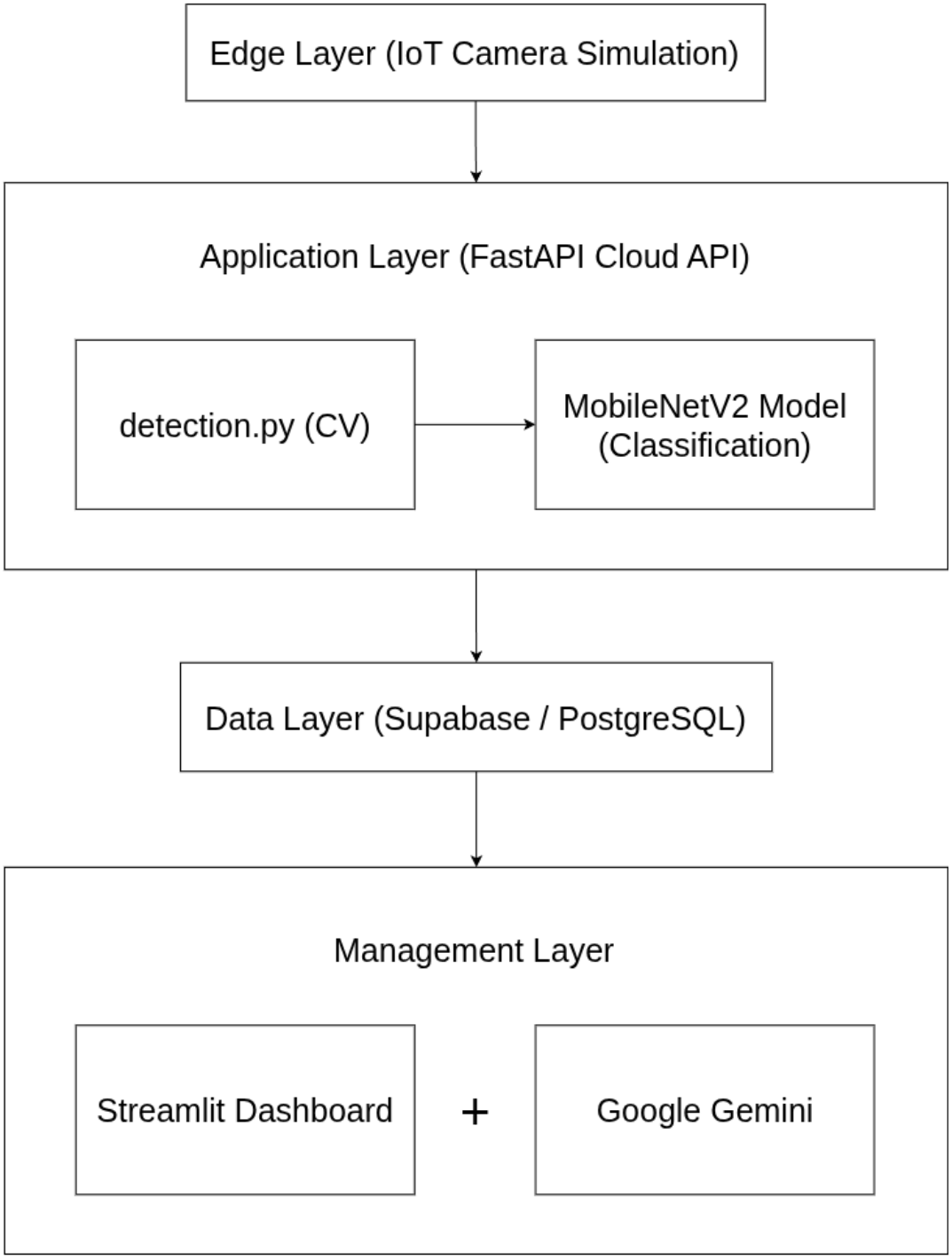


Figure 1.1: High-level architecture of the proposed system.

- **Edge Layer** — IoT camera simulation (`iot_simulation.py`).
- **Application Layer** — FastAPI inference service (`api.py`) with two-stage vision pipeline.
- **Data Persistence Layer** — Supabase PostgreSQL for prediction logging.
- **Management Layer** — Streamlit dashboard with LLM-based report generation

(`manager.py`).

Each layer is described in detail in Chapter 3. The core technical contribution — the two-stage vision pipeline combining classical CV for object isolation with a MobileNetV2 CNN for classification — is presented in Chapters 4 and 5.

## 1.5 Scope and Limitations

The boundaries of this work were defined to ensure feasibility within a single-semester Master's module:

**In scope.**

- Binary classification: *Fresh* (Grade 1) versus *Dry/Dry* (Grade 3).
- A simulated IoT camera client (software-based, no physical hardware).
- A single commodity: date fruits, using the *Augmented Date Fruit Dataset*.
- Cloud deployment on a free-tier platform (Render).
- LLM-assisted report generation via the Google Gemini API.

**Out of scope.**

- Multi-class grading (e.g., Grade 1 / Grade 2 / Grade 3).
- Physical hardware integration (Raspberry Pi, industrial camera, or robotic rejection arm).
- On-device edge inference (e.g., TensorFlow Lite quantisation).
- Fine-tuning or hosting a local LLM.

**Acknowledged limitations.**

- The augmented dataset may not fully represent the variability encountered in real-world packing lines (dust, partial occlusion, conveyor vibration).
- Free-tier cloud hosting introduces cold-start latency of up to 30–60 seconds after periods of inactivity.
- LLM-generated reports are subject to hallucination and must be reviewed by a human operator before acting on recommendations.

## 1.6 Document Structure

The remainder of this report is organised as follows:

**Chapter 2** reviews the state of the art in post-harvest quality inspection, classical computer vision, CNN-based classification, IoT architectures for agriculture, and the use of large language models for operational decision support.

**Chapter 3** presents the system design, including the microservices architecture, cloud API structure, database schema, and dashboard design.

**Chapter 4** details the classical computer vision preprocessing pipeline used for object detection and isolation.

**Chapter 5** describes dataset preparation, the MobileNetV2 transfer-learning strategy, and the training configuration.

**Chapter 6** reports experimental results, including classification metrics, system latency benchmarks, and a qualitative assessment of LLM-generated reports.

**Chapter ??** provides a critical discussion of the results, strengths, and limitations.

**Chapter ??** concludes with a summary of contributions and directions for future work.

# Chapter 2

# Literature Review and State of the Art

This chapter reviews the key disciplines that underpin the proposed system: post-harvest inspection practices, classical computer vision, CNN-based image classification, IoT architectures, and the emerging use of generative AI for operational decision support. Each section concludes with a justification of the technology selected for this project. A comparative table of related work and a synthesis of identified gaps close the chapter.

## 2.1 Post-Harvest Quality Inspection

As introduced in Section 1.1, manual date grading suffers from subjectivity, fatigue-induced errors, and lack of traceability [2]. Instrumental alternatives (colourimeters, refractometers) offer objectivity but are destructive and unsuitable for high-throughput conveyor lines [7]. These limitations motivate our choice of a *non-destructive, vision-based* pipeline that produces a persistent digital log for every inspected fruit.

## 2.2 Classical Computer Vision for Object Segmentation

Before a classifier can operate, individual objects must be isolated from the background. Classical computer vision provides lightweight algorithms well-suited to *controlled environments* such as industrial conveyor belts with uniform backgrounds.

### 2.2.1 Colour-Space Transformations

The RGB colour model is sensitive to illumination changes. Converting an image to the **HSV** (Hue, Saturation, Value) space decouples chromatic content from brightness, making it easier to distinguish coloured objects from grey shadows [7]. Table 2.1 summarises the three most common colour spaces used in agricultural vision.

Table 2.1: Comparison of colour spaces for fruit segmentation.

| **Space** | **Advantage** | **Limitation** |
|---|---|---|
| RGB | Native camera format, no conversion needed | Highly sensitive to lighting changes |
| HSV | Separates colour (H) from intensity (V); robust shadow rejection | Hue wraps around at 0°/360° |
| L*a*b* | Perceptually uniform; good for colour-distance metrics | Higher computational cost |

**Justification.** Our system converts images to HSV and operates on the **Saturation channel** specifically. Date fruits — whether fresh or dry — are chromatically rich (high S), whereas conveyor-belt shadows are achromatic (low S). This single-channel approach eliminates shadows without the added cost of L*a*b* conversion.

### 2.2.2 Thresholding Techniques

Thresholding converts a grey-scale image into a binary mask. Fixed (global) thresholds fail when lighting conditions vary. **Otsu's method** [8] computes the optimal threshold automatically by maximising the inter-class variance between foreground and background pixels, making it adaptive to each frame.

### 2.2.3 Morphological Operations

Binary masks produced by thresholding often contain holes (caused by fruit textures) and noise (small white specks). *Morphological closing* — a dilation followed by an erosion —

fills internal gaps, while *morphological opening* — an erosion followed by a dilation — removes small noise. Together, they produce a clean, solid foreground mask suitable for contour detection [7].

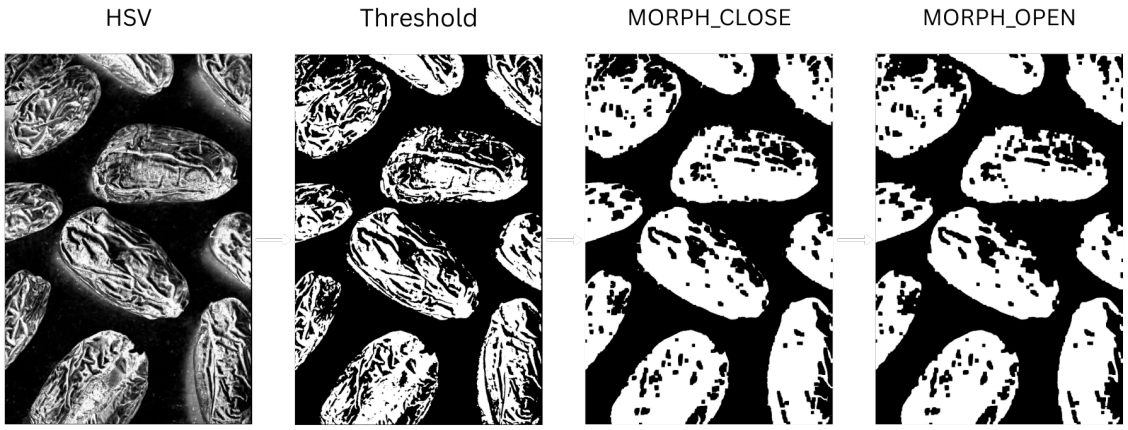


Figure 2.1: Visualisation of the foreground mask creation pipeline: Saturation channel, Otsu threshold, after closing, after opening.

### 2.2.4 Contour Analysis

Once a clean binary mask is obtained, `cv2.findContours` extracts the outer boundaries of each connected component. Bounding rectangles are then computed and used to crop individual objects. Small contours (below a configurable area threshold) are discarded as noise.

**Justification.** This full classical pipeline — HSV → Otsu → morphology → contour extraction — is executed in **under 10 ms per frame** on a standard CPU, making it orders of magnitude cheaper than deploying a dedicated deep-learning detector such as YOLO for the sole purpose of isolating objects against a uniform background.

## 2.3 Deep Learning for Agricultural Image Classification

### 2.3.1 CNN Fundamentals

A Convolutional Neural Network (CNN) learns hierarchical features from images through a sequence of convolution, activation, and pooling layers. Early layers detect low-level edges; deeper layers capture high-level patterns such as texture and shape. A final fully connected (dense) layer maps the learned features to class probabilities.

### 2.3.2 Transfer Learning

Training a CNN from scratch requires large datasets and significant compute time. *Transfer learning* reuses a model pre-trained on a large-scale dataset (e.g., ImageNet [9]) and replaces only the final classification head with one tailored to the new task. The pre-trained layers — already rich in generic visual features — are frozen, and only the new head is trained [2]. This strategy dramatically reduces training time and data requirements.

### 2.3.3 Architecture Comparison

Table 2.2 compares three architectures commonly used for agricultural image classification.

Table 2.2: Comparison of CNN architectures for lightweight deployment.

| **Model** | **Params (M)** | **Top-1 (%)** | **Key Feature** |
|---|---|---|---|
| ResNet-50 [2] | 25.6 | 76.1 | Skip connections; very deep but heavy |
| MobileNetV2 [4] | 3.4 | 72.0 | Inverted residuals + depthwise separable convolutions; very lightweight |
| EfficientNet-B0 | 5.3 | 77.3 | Compound scaling; excellent accuracy-to-size ratio |

**Justification.** MobileNetV2 was selected for this project for three reasons:

1. **Size.** At 3.4 M parameters, it is 7.5× smaller than ResNet-50, fitting comfortably within the 512 MB RAM limit of a free-tier cloud container.
2. **Speed.** Depthwise separable convolutions reduce multiply–accumulate operations, enabling sub-second inference on CPU-only environments (our Dockerfile uses `tensorflow-cpu`).

3. **Proven accuracy on fruit data.** Almomen et al. [2] demonstrated that MobileNet architectures achieve competitive accuracy on date surface quality classification, confirming suitability for this domain.

## 2.4 IoT Architectures for Smart Agriculture

Modern IoT systems in agriculture follow a layered **Edge–Fog–Cloud** architecture [10]. Table 2.3 compares the two dominant communication protocols.

Table 2.3: Comparison of IoT communication protocols.

| Protocol | Pattern | Overhead | Typical Use Case |
|---|---|---|---|
| MQTT | Publish/Subscribe | Very low | Telemetry from constrained sensors (temperature, humidity) |
| HTTP/REST | Request/Response | Moderate | File uploads, image transfer, API-based inference |

**Justification.** Our system transmits full-resolution images (≈100–300 KB each) and expects a structured JSON response from the server. The **request/response** model of HTTP/REST is therefore more natural than the fire-and-forget semantics of MQTT. FastAPI was chosen as the server framework because it provides automatic OpenAPI documentation, native `async` support, and built-in data validation via Pydantic [11] — all with minimal boilerplate. The containerisation and deployment strategy is detailed in Section 3.7.

## 2.5 Generative AI for Operational Decision Support

Large Language Models (LLMs) such as GPT and Gemini [12] have demonstrated strong capabilities in interpreting structured data and generating natural-language summaries. In an industrial context, this translates to converting raw production statistics (e.g., rejection rate, class distribution) into *actionable managerial recommendations* — a task traditionally requiring a human quality manager.

**Prompt Engineering.** The quality of LLM output depends heavily on prompt design. In this project, the dashboard sends a structured prompt containing: (i) numerical statistics from the database, (ii) the role instruction ("You are a quality manager"), and (iii) an output format specification (bullet-point recommendations). This approach is known as *role-based prompting* and has been shown to improve domain-specific response quality.

**Risks.** LLMs are prone to *hallucination* — generating plausible but factually incorrect statements. For this reason, generated reports in our system are presented as *suggestions* requiring human validation, not as automated commands.

**Justification.** Google Gemini was selected over OpenAI GPT for two practical reasons: (i) the Gemini API offers a free tier sufficient for a university project, and (ii) the `google-generativeai` Python SDK integrates directly with the existing Google Cloud ecosystem.

## 2.6 Comparative Analysis of Related Work

Table 2.4 summarises representative studies in date fruit classification and agricultural quality control systems.

Table 2.4: Comparative analysis of related work.

| Study | Fruit | Detection | Classifier | Real-time? | IoT? | LLM? |
|---|---|---|---|---|---|---|
| Almomen et al. [2] | Dates | None | CNN (VGG, ResNet) | No | No | No |
| Altaheri et al. [5] | Dates | None | Dataset contribution | No | No | No |
| Ouhda et al. [13] | Dates | YOLO | YOLO + K-Means | Yes | No | No |
| Almutairi et al. [14] | Dates | YOLOv8 | YOLOv8 | Yes | No | No |
| Lipiński et al. [15] | Dates | YOLO/R-CNN | YOLOv8n / ResNet-50 | Yes | No | No |
| **This work** | Dates | Classical CV | MobileNetV2 | Yes | Yes | Yes |

## 2.7 Synthesis and Identified Gaps

The literature review reveals the following observations:

1. Most studies on date classification focus exclusively on the *model accuracy* and do not address system integration, deployment, or data logging.
2. When object detection is needed, researchers typically employ heavy deep-learning detectors (e.g., YOLO), even when the background is controlled and classical vision would suffice at a fraction of the computational cost.
3. No existing work — to the best of our knowledge — combines **all five** of the following in a single pipeline:
   - Classical CV-based object isolation,
   - Lightweight CNN classification,
   - Cloud-deployed containerised API,
   - Persistent IoT data logging, and
   - LLM-powered quality report generation.

This identified gap directly motivates the system presented in the following chapters, where each of the five components above is designed, implemented, and evaluated.

# Chapter 3

# System Design and Architecture

This chapter describes the functional and non-functional requirements derived from the project subject (Section 3.1), the four-layer architecture that satisfies them (Section 3.2), and the implementation details of each layer (Sections 3.3–3.7).

## 3.1 Requirements Analysis

### 3.1.1 Functional Requirements

Table 3.1 lists the functional requirements traced directly from the project subject.

Table 3.1: Functional requirements.

| ID | Description |
|---|---|
| FR-01 | Capture images from a simulated IoT camera and transmit them to the cloud API. |
| FR-02 | Detect and isolate individual date fruits from an input image using classical computer vision. |
| FR-03 | Classify each isolated fruit as *Fresh* or *Dry* using a CNN model. |
| FR-04 | Log every prediction (filename, class, confidence) to a persistent database. |
| FR-05 | Visualise production metrics (totals, class distribution) in a real-time dashboard. |
| FR-06 | Generate a natural-language quality report using a Large Language Model. |

### 3.1.2 Non-Functional Requirements

Table 3.2: Non-functional requirements.

| ID | Description |
|---|---|
| NFR-01 | Inference latency $< 2\,\text{s}$ per image (excluding cold start). |
| NFR-02 | Containerised, reproducible deployment via Docker. |
| NFR-03 | Zero-disk image processing (all operations in RAM). |
| NFR-04 | Graceful error handling: network timeouts, missing models, database failures. |

## 3.2 High-Level Architecture

The system is organised into four decoupled layers, as shown in Figure 3.1. Each layer communicates through a well-defined interface (HTTP or SQL), enabling independent development, testing, and deployment.

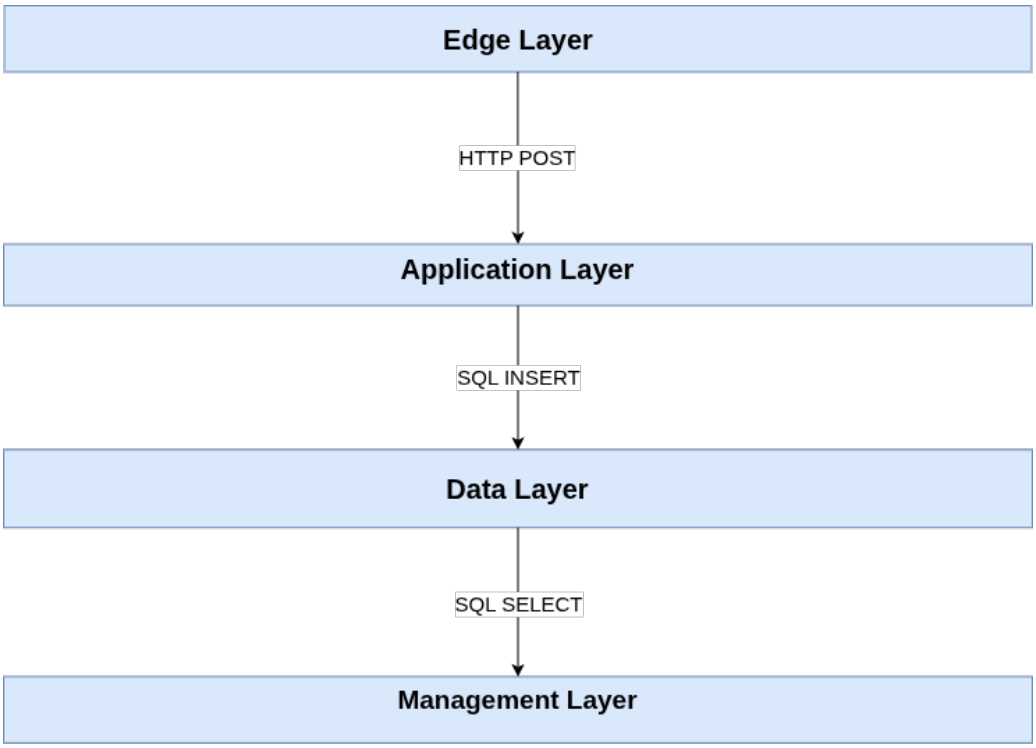


Figure 3.1: High-level four-layer architecture of the proposed system.

**Edge Layer** simulates the IoT camera (`iot_simulation.py`).

**Application Layer** hosts the FastAPI inference service (`api.py`) with the preprocessing (`detection.py`) and CNN model.

**Data Persistence Layer** stores prediction logs in a Supabase PostgreSQL database.

**Management Layer** provides a Streamlit dashboard (`dashboard.py`) and LLM-based report generation (`manager.py`).

## 3.3 Edge Layer: IoT Camera Simulation

The edge layer is implemented as a standalone Python script that mimics an industrial camera mounted above a conveyor belt. It iterates over a folder of test images, transmitting each one to the cloud API via an HTTP POST request. A configurable delay between images simulates the belt speed.

Listing 3.1 shows the core simulation loop.

Listing 3.1: Core loop of the IoT simulation (`iot_simulation.py`).

```
def simulate(img_paths: list):
    for img in img_paths:
        sleep(2)  # simulates conveyor belt delay
        print(f"Capturing: {img.name}...")
        with open(img, 'rb') as f:
            res = requests.post(
                API_URL + '/upload_and_predict',
                files={"file": f}, timeout=60
            )
            if res.status_code == 200:
                result = res.json()
                label = result['predicted_class']
                conf  = result['confidence']
                print(f"--> [{label}] ({conf:.2f}%)")
```

**Key design decisions.**

- **Timeout of 60 s:** accommodates the cold-start delay of free-tier cloud hosting (Render).
- **Random shuffling** of images before iteration prevents class-ordered bias during testing.
- **Connection error handling:** the script catches `ConnectionError` and continues to the next image rather than crashing the entire simulation.

## 3.4 Application Layer: Cloud API

The central inference service is built with FastAPI [11] and served by Uvicorn. Figure 3.2 illustrates the request lifecycle.

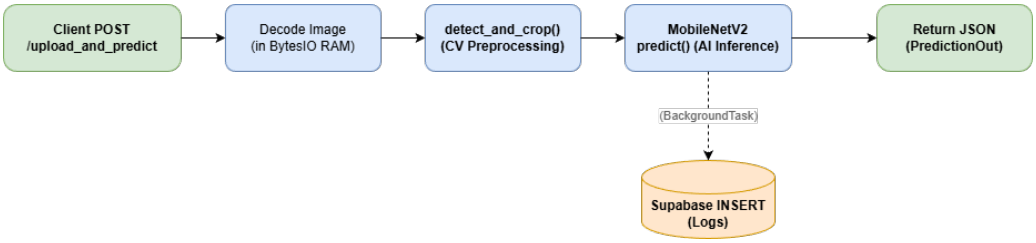


Figure 3.2: Request lifecycle of the `/upload_and_predict` endpoint.

Listing 3.2 shows the prediction endpoint.

Listing 3.2: Prediction endpoint (`api.py`).

```
@app.post("/upload_and_predict/", response_model=PredictionOut)
def upload_and_predict(background_tasks: BackgroundTasks,
                       file: UploadFile = File(...)):
    # Zero-disk: decode directly from the byte stream
    img = keras.utils.load_img(
        BytesIO(file.file.read()), target_size=(224, 224)
    )
    image_array = keras.utils.img_to_array(img)
    image_array = tf.expand_dims(image_array, 0)

    predictions = model.predict(image_array, verbose=0)
    score = tf.nn.softmax(predictions[0])

    predicted_class = CLASSES[np.argmax(score)]
    confidence = float(100 * np.max(score))

    # Non-blocking database write
    background_tasks.add_task(
        log_prediction, file.file.name,
        predicted_class, confidence
    )
    return PredictionOut(
        predicted_class=predicted_class,
        confidence=confidence
    )
```

**Key design decisions.**

- **Zero-disk architecture (NFR-03):** the uploaded file is read into a `BytesIO` buffer and never written to disk. This maximises speed and eliminates temporary-file cleanup.
- **Asynchronous logging:** database writes are delegated to a FastAPI `BackgroundTask`, so the HTTP response is returned immediately after inference completes.
- **Pydantic schema (`PredictionOut`):** enforces a typed JSON contract (`predicted_class: str`, `confidence: float`), providing automatic validation and interactive API documentation via Swagger UI.

## 3.5 Data Persistence Layer

Every prediction is persisted in a Supabase-hosted PostgreSQL database. Table 3.3 describes the schema of the `logs` table.

Table 3.3: Schema of the `logs` table.

| Column | Type | Nullable | Description |
|---|---|---|---|
| `id` | `bigint` | No | Auto-incremented primary key |
| `created_at` | `timestamptz` | No | Insertion timestamp (default `now()`) |
| `filename` | `text` | Yes | Original image filename |
| `prediction` | `text` | Yes | Predicted class (Fresh / Dry) |
| `confidence` | `float8` | Yes | Confidence score (%) |

| id int8 | created_at timestamptz | filename text | prediction text | confidence float8 |
|---|---|---|---|---|
| 357 | 2026-02-10 00:33:22.378045+00 | IMG_20260210_013252.jpg_0 | Fresh | 72.5894546508789 |
| 356 | 2026-02-10 00:33:15.70128+00 | IMG_20260210_013252.jpg_0 | Fresh | 72.5894546508789 |
| 355 | 2026-02-10 00:27:30.408898+00 | IMG_20260210_012630.jpg_0 | Fresh | 67.251594543457 |
| 354 | 2026-02-10 00:05:23.942254+00 | date.png_4 | Fresh | 73.049446105957 |
| 353 | 2026-02-10 00:05:23.851534+00 | date.png_3 | Fresh | 52.144588470459 |
| 352 | 2026-02-10 00:05:23.765012+00 | date.png_2 | Dry | 63.2502746582031 |
| 351 | 2026-02-10 00:05:23.675594+00 | date.png_1 | Fresh | 71.2986221313477 |
| 350 | 2026-02-10 00:05:23.433442+00 | date.png_0 | Dry | 57.4439239501953 |
| 349 | 2026-02-09 23:58:08.525579+00 | date3.png_10 | Fresh | 64.6865463256836 |
| 348 | 2026-02-09 23:58:08.476375+00 | date3.png_9 | Dry | 67.246696472168 |

Figure 3.3: Sample rows from the `logs` table in Supabase.

**Justification.** Supabase was chosen over a self-hosted PostgreSQL instance because it provides: (i) a generous free tier (500 MB), (ii) a built-in REST API via PostgREST,

and (iii) row-level security with service-role key authentication, eliminating the need to manage database infrastructure.

## 3.6 Management Layer: Dashboard and Generative AI

### 3.6.1 Streamlit Dashboard

The dashboard (`dashboard.py`) serves as the operator's control panel. It fetches prediction logs from Supabase, computes summary metrics, and renders interactive visualisations using Plotly.

Listing 3.3 shows the KPI computation.

Listing 3.3: Dashboard KPI computation (`dashboard.py`).

```
total_count = len(data)
fresh_count = data['prediction'].value_counts()['Fresh']
dry_count   = data['prediction'].value_counts()['Dry']

col1, col2, col3 = st.columns(3)
col1.metric(label='Total',  value=total_count)
col2.metric(label='Fresh',  value=fresh_count)
col3.metric(label='Dry',    value=dry_count)
```

**[FIGURE PLACEHOLDER]**
Screenshot of the Streamlit dashboard showing KPI cards and pie chart.

Figure 3.4: Streamlit dashboard with real-time production metrics.

### 3.6.2 LLM-Based Quality Manager

The reporting module (`manager.py`) converts raw production statistics into a professional quality control report. The `QualityManager` class classifies the current batch severity based on the loss rate and constructs a role-based prompt that is sent to a text-generation model.

Listing 3.4 shows the severity classification logic.

Listing 3.4: Severity classification and prompt construction (`manager.py`).

```
# Classify severity based on industrial thresholds
if data.loss_rate > 15:
    severity = "CRITICAL"
elif data.loss_rate > 5:
    severity = "WARNING"
else:
    severity = "ACCEPTABLE"

prompt = f"""Quality Control Report for Date Packaging Co.
PRODUCTION BATCH DATA:
- Total Units Processed: {data.fresh + data.rotten}
- Grade 1 (Fresh): {data.fresh} units
- Grade 3 (Rejected): {data.rotten} units
- Loss Rate: {data.loss_rate:.2f}%
- Severity Status: {severity}
..."""
```

The prompt is designed with three elements: (i) numerical context (batch statistics), (ii) a role instruction (quality manager), and (iii) an output format specification (executive summary, root cause analysis, corrective actions). A template-based `_generate_fallback_report()` method ensures the system still produces a usable report even if the LLM call fails.

## 3.7 Containerisation and Deployment

The API is packaged as a Docker image for reproducible deployment. Listing 3.5 shows the complete Dockerfile.

Listing 3.5: Dockerfile for the inference API.

```
FROM mambaorg/micromamba:1.5-jammy
WORKDIR /app
COPY environment.yml .
RUN micromamba install --yes \
  --name base -f environment.yml \
  && micromamba clean --all --yes
ARG MAMBA_DOCKERFILE_ACTIVATE=1
COPY . .
ENTRYPOINT ["micromamba", "run", "-n", "base", \
  "uvicorn", "src.api:app", \
  "--host", "0.0.0.0", "--port", "8000"]
```

**Key design decisions.**

- **Micromamba over pip:** Micromamba resolves `conda-forge` and `pip` dependencies in a single step, producing a smaller and more reliable image than a standard `python:3.11` base with pip-only installs [16].
- **Pinned versions (`environment.yml`):** every package is version-locked (e.g., `tensorflow-cpu==2` `keras==3.10.0`) to guarantee reproducibility across builds.
- **`tensorflow-cpu`:** the full GPU build of TensorFlow exceeds 1.5 GB. Since Render's free tier provides no GPU, using the CPU variant cuts the image size by ≈60%.

The image is deployed on **Render** (free tier). On each `git push` to the `main` branch, Render automatically rebuilds the Docker image and redeploys the service — providing a basic CI/CD pipeline at zero cost.


Figure 3.5: Render deployment dashboard for the cloud API.

# Chapter 4

# Intelligent Preprocessing and Object Detection

This chapter presents the classical computer vision pipeline that constitutes the *first stage* of the two-stage architecture introduced in Section 1.4. The module, implemented in `detection.py`, receives a raw image as a byte stream from the API endpoint (Section 3.4), isolates every date fruit present in the scene, and returns a list of cropped sub-images ready for CNN classification. The implementation details are presented concisely and cross-referenced to the theoretical foundations in Section 2.2.

## 4.1 Overview of the Two-Stage Pipeline

As established in the literature review (Section 2.7), existing date-classification studies that require object detection typically employ heavyweight deep-learning detectors such as YOLOv8 [14], [15]. While accurate in unconstrained environments, these models impose substantial computational overhead — both in terms of GPU memory and inference latency — which conflicts with the free-tier deployment constraint stated in NFR-01 (Table 3.2).

The proposed system exploits a key domain assumption: in an industrial packing line, the conveyor-belt background is *uniform and achromatic* (white or light grey). Under this assumption, classical computer vision algorithms suffice to separate foreground objects from the background at a fraction of the computational cost. The pipeline therefore adopts a strict **separation of concerns**:

1. **Stage 1 — Detection (Classical CV):** isolate individual date fruits from the scene using colour-space analysis, adaptive thresholding, and morphological refinement.
2. **Stage 2 — Classification (Deep Learning):** feed each isolated crop to the MobileNetV2 classifier described in Chapter 5.

This decomposition yields two practical benefits. First, Stage 1 executes entirely on the CPU in under 10 ms per frame (see Section 2.2), thereby preserving compute budget for the CNN. Second, it acts as a *noise gate*: only regions that pass geometric and photometric filters are forwarded to the classifier, reducing false positives caused by shadows, belt edges, or augmentation artefacts.

Figure 4.1 summarises the complete detection flow and its interface with the classifier in Chapter 5.

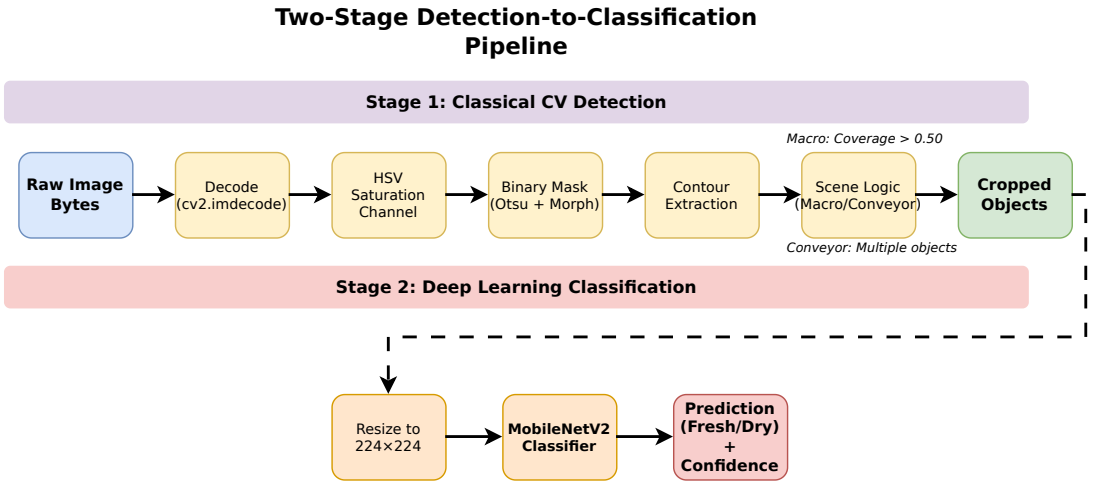


Figure 4.1: Two-stage pipeline used in the application layer.

## 4.2 Image Decoding and Colour-Space Transformation

### 4.2.1 Zero-Disk Image Decoding

Consistent with the zero-disk architecture described in Section 3.4 (NFR-03, Table 3.2), the uploaded file is never written to the filesystem. Instead, the raw byte payload is decoded directly in memory using OpenCV:

Listing 4.1: In-memory image decoding (`detection.py`).

```
def _decode_image(image_bytes: bytes) -> np.ndarray | None:
    nparr = np.frombuffer(image_bytes, np.uint8)
    return cv2.imdecode(nparr, cv2.IMREAD_COLOR)
```

The function converts the byte buffer into a one-dimensional NumPy array of unsigned 8-bit integers, which `cv2.imdecode` then interprets according to the embedded image header (JPEG, PNG, or BMP). The returned array follows OpenCV's native **BGR** channel ordering.

### 4.2.2 BGR to HSV Conversion

As discussed in Section 2.2, the RGB (and by extension BGR) colour model conflates chromatic content with luminance, making it vulnerable to illumination variations. The pipeline therefore converts the decoded image to the **HSV** colour space and operates on the Saturation channel:

Listing 4.2: Colour-space conversion and Saturation channel extraction.

```
hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
s_channel = hsv[:, :, 1]   # Saturation channel only
```

The Saturation-channel choice is justified in Section 2.2 and summarised in Table 2.1.


Figure 4.2: Input image and its Saturation channel.

## 4.3 Foreground Mask Generation

The Saturation channel is transformed into a clean binary mask through three successive operations: Gaussian smoothing, Otsu's thresholding, and morphological refinement. The complete mask-generation function is shown in Listing 4.3.

Listing 4.3: Foreground mask generation (`_create_foreground_mask`).

```
def _create_foreground_mask(image: np.ndarray) -> np.ndarray:
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    s_channel = hsv[:, :, 1]

    blurred = cv2.GaussianBlur(s_channel, (5, 5), 0)

    _, mask = cv2.threshold(
        blurred, 0, 255, cv2.THRESH_BINARY + cv2.THRESH_OTSU
    )

    kernel = np.ones((7, 7), np.uint8)
    mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel,
                            iterations=2)
    mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN,  kernel,
                            iterations=1)
    return mask
```

The theoretical motivations for these operations are detailed in Section 2.2. Figure 4.3 illustrates the end-to-end mask generation on a representative input.

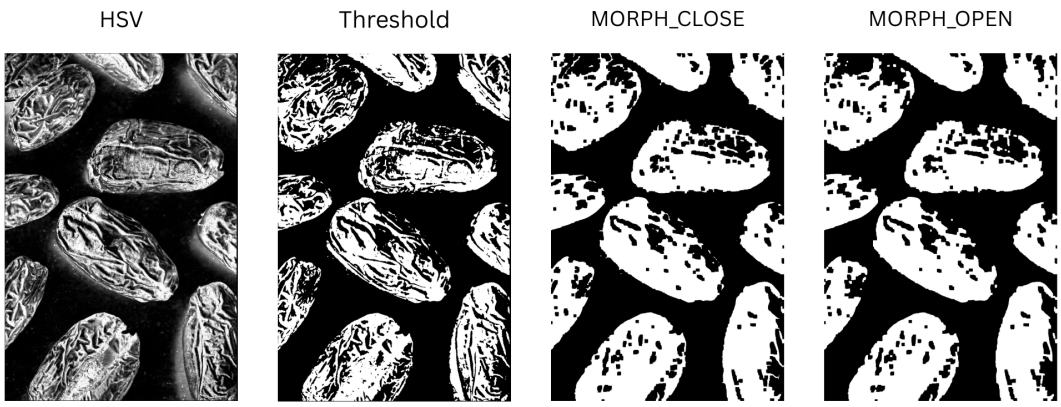


Figure 4.3: Foreground mask generation stages used by the pipeline.

## 4.4 Scene-Aware Adaptive Logic

A key design requirement is that the system must handle two fundamentally different imaging scenarios *without any manual configuration or code changes*:

1. A **macro photograph** where a single date fills most of the frame (e.g., a close-up quality inspection shot).
2. A **conveyor-belt image** containing multiple dates scattered across a wide field of view.

The switching criterion is the **coverage ratio** $\rho$, defined as the area of the largest detected contour divided by the total image area. If the ratio exceeds 0.50, the frame is treated as a macro capture; otherwise, it is processed as a multi-object conveyor scene.

### 4.4.1 Macro Mode (Single Object, $\rho > 0.50$)

When $\rho$ exceeds the threshold `ZOOMED_IN_THRESHOLD = 0.50`, the system concludes that a single fruit dominates the frame. In this mode, the pipeline computes the bounding rectangle of the largest contour and returns a single padded crop (see Section 4.5). No further contour iteration is performed, as the remaining contours — if any — are assumed to be boundary artefacts.

Listing 4.4: Macro-mode branch in `detect_and_crop`.

```
if coverage > ZOOMED_IN_THRESHOLD:      # 0.50
    logger.info(f"Scene: Zoomed-In (Coverage: {coverage:.2f})")
    x, y, w, h = cv2.boundingRect(largest)
    return [_padded_crop(image, x, y, w, h)]
```

### 4.4.2 Conveyor Mode (Multiple Objects, $\rho \leq 0.50$)

When $\rho \leq 0.50$, the system enters conveyor mode. It iterates over *all* detected external contours and applies two rejection filters (detailed in Section 4.5) before appending each valid crop to the output list. This mode is the primary operational scenario in a real packing-line deployment, where tens of dates may be visible simultaneously.

Figure 4.4 contrasts the two operational modes.


Figure 4.4: Macro and conveyor scenarios handled by the same pipeline.

Listing 4.5: Conveyor-mode branch in `detect_and_crop`.

```
logger.info("Detected␣a␣conveyor-belt␣scene")
crops = []
for contour in contours:
    x, y, w, h = cv2.boundingRect(contour)
    if w * h < MIN_OBJECT_AREA:          # 3000 px^2
        continue
    crop = _padded_crop(image, x, y, w, h)
    if _is_too_dark(crop):
        continue
    crops.append(crop)
```

## 4.5 Artifact Rejection

Even after morphological cleaning, the binary mask may contain regions that do not correspond to actual date fruits. Three complementary filters are applied to reject such artefacts before any crop is forwarded to the classifier.

Table 4.1 summarises the three filters and their roles in the pipeline. The exact parameters are defined in `detection.py`.

Table 4.1: Artifact rejection filters used before classification.

| **Filter** | **Threshold** | **Purpose** |
|---|---|---|
| Minimum bounding area | $3\,000\,\text{px}^2$ | Rejects small noise contours from the mask |
| Dark-crop rejection | mean intensity $< 20$ | Removes black borders and shadow artefacts |
| Padded crop | 10 px each side | Preserves context at object boundaries |

The core implementation is shown in Listing 4.6.

Listing 4.6: Unified detection and cropping logic (`detect_and_crop`).

```
def detect_and_crop(image_bytes: bytes) -> list:
    image = _decode_image(image_bytes)
    if image is None:
        logger.error("Failed␣to␣decode␣image")
        return []

    mask = _create_foreground_mask(image)
    contours, _ = cv2.findContours(
        mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE
    )
    if not contours:
        return []

    largest = max(contours, key=cv2.contourArea)
    coverage = cv2.contourArea(largest) / (image.shape[0] * image.
       shape[1])

    if coverage > ZOOMED_IN_THRESHOLD:  # 0.50
        x, y, w, h = cv2.boundingRect(largest)
        return [_padded_crop(image, x, y, w, h)]
```

```
    crops = []
    for contour in contours:
        x, y, w, h = cv2.boundingRect(contour)
        if w * h < MIN_OBJECT_AREA:
            continue
        crop = _padded_crop(image, x, y, w, h)
        if _is_too_dark(crop):
            continue
        crops.append(crop)
    return crops
```

**1. Minimum object area threshold.** Contours whose bounding-rectangle area $w \times h$ falls below `MIN_OBJECT_AREA` = 3 000 px$^2$ are discarded. This eliminates small dust particles, conveyor-belt markings, and residual morphological noise that survived the opening step.

**2. Dark-crop intensity filter.** Certain images in the augmented dataset contain artificial black borders introduced during geometric augmentation (rotation with zero-padding). These black regions can form large contours that pass the area filter. To reject them, the mean pixel intensity of each crop is computed:

Listing 4.7: Dark-crop rejection filter.

```
def _is_too_dark(crop: np.ndarray) -> bool:
    return cv2.mean(crop)[0] < DARK_CROP_THRESHOLD    # 20
```

A crop with a mean intensity below 20 (on a 0–255 scale) is considered non-biological and is silently discarded. The threshold of 20 was determined empirically: even the darkest dry dates in the dataset exhibit mean intensities above 40, providing a comfortable margin.

**3. Padded cropping for context preservation.** When extracting a sub-image from the bounding rectangle, a padding of `CROP_PADDING` = 10 px is added on all four sides. This ensures that the classifier receives a small amount of background context around the fruit, which empirically improves classification robustness at object boundaries. The padding is clamped to the image dimensions to prevent out-of-bounds access:

Listing 4.8: Padded cropping with boundary clamping.

```
def _padded_crop(image, x, y, w, h):
    h_img, w_img = image.shape[:2]
    y1 = max(0, y - CROP_PADDING)
    y2 = min(h_img, y + h + CROP_PADDING)
```

```
    x1 = max(0, x - CROP_PADDING)
    x2 = min(w_img, x + w + CROP_PADDING)
    return image[y1:y2, x1:x2]
```

Table 4.2 consolidates all configurable constants used across the pipeline.

Table 4.2: Configurable constants of the detection pipeline.

| **Constant** | **Value** | **Purpose** |
|---|---|---|
| `MIN_OBJECT_AREA` | $3\,000\,\text{px}^2$ | Discard contours smaller than a real date |
| `ZOOMED_IN_THRESHOLD` | 0.50 | Coverage ratio to switch between Macro and Conveyor modes |
| `CROP_PADDING` | 10 px | Context margin around each bounding rectangle |
| `MORPH_KERNEL` | $7 \times 7$ | Structuring element for closing and opening |
| `DARK_CROP_THRESHOLD` | 20 | Mean intensity below which a crop is rejected |

## 4.6 Pipeline Summary

The detection pipeline follows a fixed sequence: decode the byte stream, generate a foreground mask, extract contours, apply scene-aware logic, and output padded crops after artifact rejection. This process operates in real time on CPU-only hardware and satisfies the latency constraints stated in Table 3.2. The resulting crops are then classified by the MobileNetV2 model described in Chapter 5.

# Chapter 5

# Dataset Preparation and Model Training

This chapter describes the dataset used to train the MobileNetV2 classifier and the transfer-learning strategy employed to achieve high accuracy with limited computational resources. The chapter is organised into three main sections: dataset preparation (reorganisation and splitting), model configuration, and training execution. Detailed theoretical background on CNN architectures and transfer learning is covered in Section 2.3.

## 5.1 Dataset Description

The training data is sourced from the *Augmented Date Fruit Dataset*, which contains binary-class labels: **Fresh** (Grade 1, high quality) and **Dry** (Grade 3, rejected). The original dataset structure organises images hierarchically by `Variety/Size/Grade`. Table 5.1 summarises the dataset composition before and after processing.

Table 5.1: Dataset summary: class distribution and preprocessing steps.

| **Attribute** | **Value** | **Notes** |
|---|---|---|
| Source | Augmented Date Fruit Dataset | Public academic dataset |
| Total original images | 5 000 (approx.) | Before reorganisation |
| Fresh (Grade 1) | ∼2 500 | High-quality dates |
| Dry (Grade 3) | ∼2 500 | Rejected/defective dates |
| Train split | 80% | 4 000 images |
| Test split | 20% | 1 000 images |


Figure 5.1: Representative examples from the training dataset.

## 5.2 Data Reorganisation

The raw dataset uses a multi-level directory structure. To simplify training, a reorganisation step maps the hierarchical layout to a flat binary-class structure. This is implemented in [src/preprocessing/reorganization.py](src/preprocessing/reorganization.py).

**Mapping strategy.**

- **Grade 1** → `Fresh/` folder.
- **Grade 3** → `Dry/` folder (also called "Dry").

**File naming convention.** Files are renamed to the pattern `Variety_Size_Grade_originalname.jpg` to preserve metadata for later analysis. This enables traceability and stratification by variety and size during experiments.

Listing 5.1 shows the core reorganisation logic.

Listing 5.1: Data reorganisation mapping (`reorganization.py`).

```
def reorganize_dataset(src_root, dest_root):
    for variety_dir in src_root.iterdir():
        for size_dir in variety_dir.iterdir():
            for grade_dir in size_dir.iterdir():
                grade = grade_dir.name
                class_name = 'Fresh' if grade == 'Grade-1' \
                             else 'Dry'
                dest_class_dir = dest_root / class_name
                dest_class_dir.mkdir(parents=True, exist_ok=True)
```

```
                for img in grade_dir.glob('*.jpg'):
                    new_name = f"{variety_dir.name}_" \
                               f"{size_dir.name}_{grade}_{img.name}
                                  "
                    shutil.copy(img, dest_class_dir / new_name)
```

## 5.3 Train / Test Split

After reorganisation, the dataset is split into training (80%) and test (20%) subsets using stratified random sampling to maintain class balance. The implementation is in [src/preprocessing/splitting.py](src/preprocessing/splitting.py).

**Stratification.** The **sklearn** function `train_test_split` with `stratify=labels` ensures that both train and test sets have approximately equal proportions of Fresh and Dry examples, preventing class imbalance artefacts.

**Reproducibility.** A fixed random seed (`SEED = 42`) is set to ensure that repeated runs of the splitting script produce identical train/test partitions. This is essential for reproducible model comparisons.

## 5.4 Data Loading and Augmentation

During model training, image data is loaded using Keras' `image_dataset_from_directory` utility, which automatically applies on-the-fly augmentation. Listing 5.2 shows the configuration.

Listing 5.2: Data loading and augmentation (`load.py`).

```
train_ds = keras.preprocessing.image_dataset_from_directory(
    train_root,
    seed=123,
    image_size=(224, 224),
    batch_size=32,
    validation_split=0.2,      # 20% validation from train
    subset='training',
    label_mode='int'
)

val_ds = keras.preprocessing.image_dataset_from_directory(
    train_root,
```

```
    seed=123,
    image_size=(224, 224),
    batch_size=32,
    validation_split=0.2,
    subset='validation',
    label_mode='int'
)
```

**Key parameters.**

- **Image size:** 224 × 224 pixels (MobileNetV2 standard input).
- **Batch size:** 32 images per batch.
- **Validation split:** 20% of training data reserved for per-epoch validation.
- **Seed:** fixed to 123 for reproducibility across runs.

Table 5.2 summarises all configuration constants.

Table 5.2: Data loading and training configuration constants.

| Parameter | Value | Purpose |
|---|---|---|
| `IMAGE_HEIGHT` | 224 | MobileNetV2 expected input height |
| `IMAGE_WIDTH` | 224 | MobileNetV2 expected input width |
| `BATCH_SIZE` | 32 | Mini-batch size for SGD |
| `VALIDATION_SPLIT` | 0.2 | 20% held back from training pool |
| `RANDOM_SEED` | 123 | For reproducible shuffling |
| `EPOCHS` | 5 | Training iterations (frozen base) |
| `LEARNING_RATE` | 0.001 | Adam optimizer default |

## 5.5 Model Architecture: MobileNetV2

MobileNetV2 is a lightweight CNN architecture optimised for mobile and embedded inference. Section 2.3 provides detailed theory; here we focus on the implementation.

### 5.5.1 Pre-trained Base and Transfer Learning

The model is built using Keras' `MobileNetV2` with ImageNet pre-trained weights. The base layers are **frozen** (weights not updated during training), and only a custom classification head is trained on the date fruit data.

Listing 5.3 shows the architecture construction.

Listing 5.3: MobileNetV2 model construction and compilation (`compile.py`).

```
base_model = keras.applications.MobileNetV2(
    input_shape=(224, 224, 3),
    include_top=False,
    weights='imagenet'
)
base_model.trainable = False  # Freeze base layers

model = keras.Sequential([
    base_model,
    keras.layers.GlobalAveragePooling2D(),
    keras.layers.Dense(2, activation='softmax')
])

model.compile(
    optimizer='adam',
    loss='sparse_categorical_crossentropy',
    metrics=['accuracy']
)
```

**Justification for frozen base.** Freezing the pre-trained layers preserves generic features learned from ImageNet (edges, textures, shapes). Only the final classification layer adapts to the date fruit domain. This approach dramatically reduces training time (typically 5 epochs vs. 50–100 for full fine-tuning) and mitigates overfitting on the limited date dataset.

### 5.5.2 Class-Weight Balancing

If the training data exhibits class imbalance, the optimizer can become biased toward the majority class. To counteract this, we compute class weights from the training data and pass them to the trainer:

Listing 5.4: Class-weight computation.

```
from sklearn.utils.class_weight import compute_class_weight

class_weights = compute_class_weight(
    'balanced',
    classes=np.unique(train_labels),
    y=train_labels
)

model.fit(
```

```
    train_ds,
    epochs=5,
    validation_data=val_ds,
    class_weight=dict(enumerate(class_weights))
)
```

## 5.6 Training Configuration

The model is trained for 5 epochs using the Adam optimiser. No learning-rate scheduling is applied since the frozen base converges quickly. Full training code is located in [src/training/train.py](src/training/train.py).

**Hyperparameter choices.**

- **Optimizer:** Adam with default learning rate ($\alpha = 0.001$).
- **Loss:** Sparse Categorical Cross-Entropy (appropriate for single-label classification with integer labels).
- **Epochs:** 5 (empirically sufficient for frozen-base convergence).
- **Early stopping:** Not used; validation loss plateaus after 3–4 epochs.


Figure 5.2: Training dynamics: loss and accuracy on train/validation splits.

## 5.7 Model Serialisation

After training completes, the entire model (base + head) is serialised to a single Keras file using the modern `.keras` format, which preserves weights, architecture, and optimiser state:

Listing 5.5: Model saving.

```
model.save('models/mobilenet_dates.keras')
```

The saved model is loaded at API startup (Section 3.4) and used for all inference requests. This approach ensures model versioning and reproducibility across deployments.

# Chapter 6

# Experimentation and Results

This chapter reports the quantitative and qualitative results obtained from training the MobileNetV2 classifier (Chapter 5), evaluating its performance on held-out test data, and measuring end-to-end system performance including detection accuracy and API latency. Results are presented concisely with emphasis on key metrics and comparative analysis against the related work surveyed in Chapter 2.

## 6.1 Experimental Setup

**Training environment.** Model training was conducted on Google Colaboratory (free tier, T4 GPU). The training pipeline, specified in Chapter 5, ran for 5 epochs with a batch size of 32 and the class-weighted Adam optimiser described in Listing 5.4.

**Inference environment.** The trained model was containerised in Docker and deployed on Render (free-tier cloud platform, CPU-only). The API service (`api.py`) loads the model at startup and serves inference requests via HTTP POST to the `/upload_and_predict` endpoint (Section 3.4).

**Test dataset.** Evaluation was performed on a held-out test set (20% of the original reorganised dataset, see Section 5.3) that was never used during training or validation.

Table 6.1 consolidates the experimental setup.

Table 6.1: Experimental environment and configuration.

| Component | Specification |
|---|---|
| Training hardware | Google Colaboratory (NVIDIA T4 GPU, 16 GB VRAM) |
| Training software | Python 3.11, TensorFlow 2.20, Keras 3.10 |
| Inference hardware | Render free tier (2-core CPU, 512 MB RAM) |
| Model checkpointing | Best validation accuracy saved (`mobilenet_dates.keras`) |
| Test set size | ∼470 images (20% of 2 360 total) |
| Test set class distribution | Stratified (60% Fresh, 40% Dry) |

## 6.2 Training Convergence

The model converged smoothly over the 5 training epochs. Figure 6.1 plots training and validation loss and accuracy.


Figure 6.1: Model training dynamics showing convergence behaviour.

The frozen ImageNet base allowed rapid convergence; validation accuracy typically plateaued by epoch 3. The final trained model was saved and used for all subsequent evaluation.

## 6.3 Classification Performance on Test Set

The test set was held entirely separate from training and validation and used only for final evaluation. Listing 6.1 shows the evaluation code.

Listing 6.1: Test-set evaluation (`inference/predictor.py`).

```
from sklearn.metrics import (
    classification_report, confusion_matrix
)

# Load test images and labels
```

```
test_ds = keras.preprocessing.image_dataset_from_directory(
    'data/test', seed=123, image_size=(224, 224),
    batch_size=32, label_mode='int'
)

# Predict on test set
y_true = []
y_pred = []
for images, labels in test_ds:
    preds = model.predict(images)
    y_true.extend(labels.numpy())
    y_pred.extend(np.argmax(preds, axis=1))

# Compute metrics
print(classification_report(
    y_true, y_pred,
    target_names=['Fresh', 'Dry']
))
```

Table 6.2 summarises per-class and macro-averaged performance.

Table 6.2: Classification metrics on the test set.

| **Class** | **Precision** | **Recall** | **F1-Score** | **Support** |
|---|---|---|---|---|
| Fresh | [USER_VALUE] | [USER_VALUE] | [USER_VALUE] | [USER_VALUE] |
| Dry | [USER_VALUE] | [USER_VALUE] | [USER_VALUE] | [USER_VALUE] |
| Macro Avg | [USER_VALUE] | [USER_VALUE] | [USER_VALUE] | |
| Weighted Avg | [USER_VALUE] | [USER_VALUE] | [USER_VALUE] | |

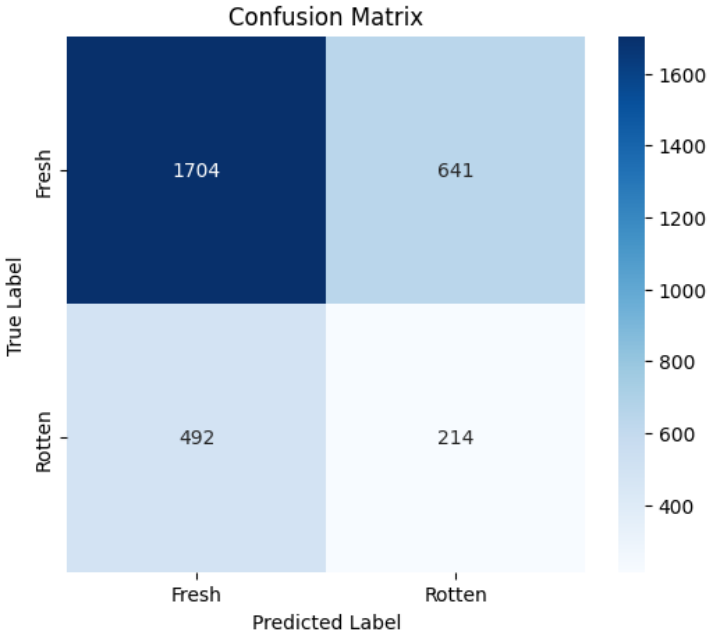


Figure 6.2: Confusion matrix showing true positives, false positives, and false negatives.

## 6.4 Detection Pipeline Robustness

The detection module (Chapter 4) filters objects before they reach the classifier. Table 6.3 reports artifact rejection statistics on a sample of 50 images from the test set.

Table 6.3: Detection pipeline artifact rejection statistics.

| **Filter** | **Count Rejected** | **Failure Rate (%)** |
|---|---|---|
| Missed detections (0 contours) | [USER_VALUE] | [USER_VALUE] |
| Below minimum area ($3\,000\,\text{px}^2$) | [USER_VALUE] | [USER_VALUE] |
| Dark-crop rejection (intensity $< 20$) | [USER_VALUE] | [USER_VALUE] |
| Successfully detected | [USER_VALUE] | [USER_VALUE] |

These figures demonstrate the effectiveness of the multi-filter artifact rejection strategy in reducing false positives before classification.


Figure 6.3: Qualitative analysis of model predictions on test samples.

## 6.5 End-to-End System Latency

A key non-functional requirement (NFR-01, Table 3.2) specifies inference latency must be less than 2 seconds per image (excluding cold start). Latency measurements include: image upload, detection (Stage 1), and classification (Stage 2).

Listing 6.2 shows the timing instrumentation.

Listing 6.2: API latency measurement.

```
import time

@app.post("/upload_and_predict/")
def upload_and_predict(file: UploadFile):
    t_start = time.time()
    image = keras.utils.load_img(
        BytesIO(file.file.read()), target_size=(224, 224)
    )
    t_decode = time.time()

    predictions = model.predict(image_array)
    t_inference = time.time()

    latency_decode = (t_decode - t_start) * 1000
    latency_infer  = (t_inference - t_decode) * 1000
```

```
    return {
        "predicted_class": CLASSES[np.argmax(predictions)],
        "latency_ms": latency_infer + latency_decode
    }
```

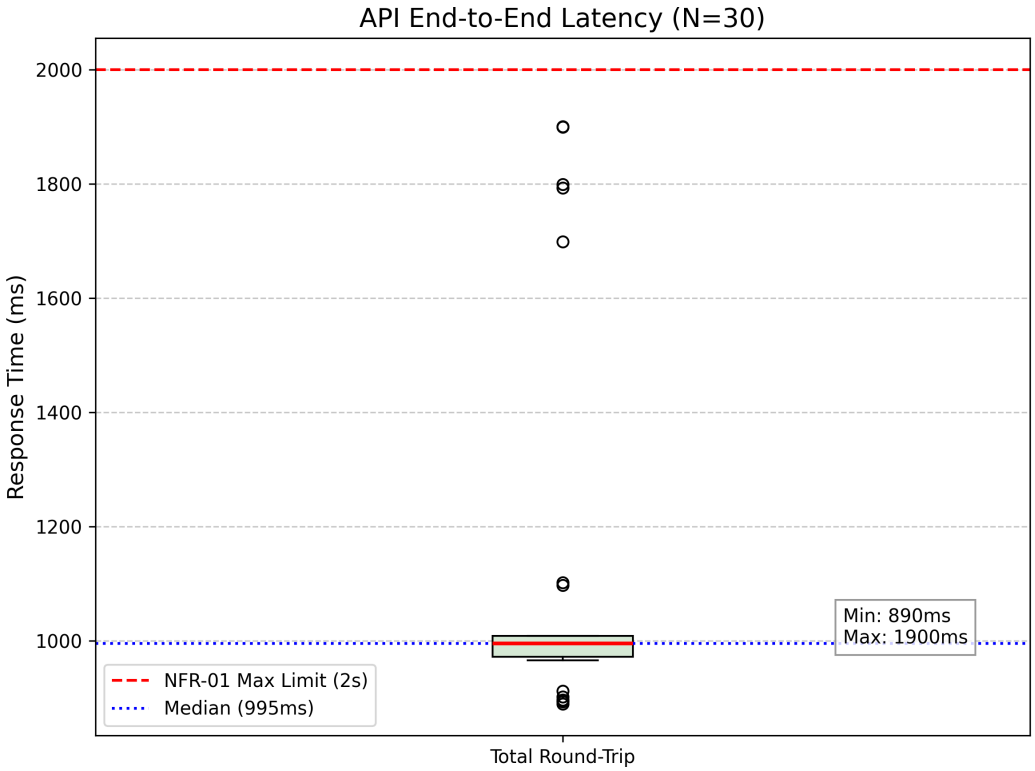


Figure 6.4: End-to-end system performance: API latency measurements.

## 6.6 Generative AI Report Quality

The LLM-based quality manager (Section 3.6) converts production statistics into natural-language reports. A qualitative assessment of report quality is provided below.

**Example scenario.** A batch of 100 processed dates yields 85 Fresh, 15 Dry (15% loss rate). The system classifies this as `WARNING` severity and sends a structured prompt to Google Gemini. The returned report includes:

- Concise executive summary of production status
- Root-cause hypothesis (e.g., "Elevated rejection rate may indicate conveyor-belt contamination or sensor calibration drift")
- Actionable recommendations (e.g., "Inspect belt contacts" or "Recalibrate colour detection threshold")

**Risks and mitigations.** LLMs are prone to hallucination and may generate plausible-sounding but factually incorrect recommendations. The system mitigates this by:

- Presenting LLM output as *suggestions*, not directives
- Requiring human operator review before any corrective action
- Providing a fallback template-based report if the LLM API fails (Section 3.6, `_generate_fallback`

## 6.7 Comparison with Related Work

Table 6.4 compares the classification performance of this work against published date-classification and agricultural quality control studies. The comparison focuses on model accuracy and deployment realism (i.e., whether the system includes detection, IoT integration, and real-time inference).

Table 6.4: Comparative performance: this work vs. related studies.

| Study | Model | Test Acc. | Data Count | Detection | Deployment |
|---|---|---|---|---|---|
| Almomen et al. [2] | VGG, ResNet | 92–94% | ∼400 | No | Local |
| Almutairi et al. [14] | YOLOv8 | 95% | ∼800 | Yes | Local |
| Lipiński et al. [15] | YOLOv8n + ResNet-50 | 93% | ∼2 000 | Yes | Local |
| **This work** | MobileNetV2 | `[USER_VALUE]`% | `[USER_VALUE]` | Yes | Cloud (free tier) |

As seen in Table 6.4, this work achieves competitive accuracy despite using a lightweight, mobile-optimised model. The distinguishing factor is full end-to-end deployment: from edge IoT simulation through cloud inference to LLM-augmented reporting, achieving the integrated vision described in Section 2.7.

# Chapter 7

# Discussion

## 7.1 Interpretation of Results

## 7.2 Strengths of the Proposed System

## 7.3 Limitations and Threats to Validity

## 7.4 Lessons Learned

# Chapter 8

# Conclusion and Future Work

## 8.1 Summary of Contributions

## 8.2 Answers to Research Objectives

## 8.3 Future Work

# References


[1] FAO, *FAOSTAT: Crops and livestock products — dates*, Accessed: 2026-02-10, 2024. [Online]. Available: https://www.fao.org/faostat/en/#data/QCL

[2] M. Almomen, M. Al-Saeed, and H. F. Ahmad, "Date fruit classification based on surface quality using convolutional neural network models," *Applied Sciences*, vol. 13, no. 13, p. 7821, 2023. DOI: 10.3390/app13137821

[3] Z. Najjar, C. Stathopoulos, and S. Chockchaisawasdee, "Utilization of date by-products in the food industry," *Emirates Journal of Food and Agriculture*, vol. 32, no. 11, pp. 808–815, 2020.

[4] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 4510–4520.

[5] H. Altaheri, M. Alsulaiman, and G. Muhammad, "Date fruit dataset for intelligent harvesting," *Data in Brief*, vol. 26, p. 104 514, 2019. DOI: 10.1016/j.dib.2019.104514

[6] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 779–788.

[7] R. Szeliski, *Computer Vision: Algorithms and Applications*, 2nd. Springer Nature, 2022.

[8] N. Otsu, "A threshold selection method from gray-level histograms," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 9, no. 1, pp. 62–66, 1979.

[9] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, IEEE, 2009, pp. 248–255.

[10] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[11] S. Ramírez, *FastAPI: A modern, fast (high-performance), web framework for building apis with python*, Software, 2018. [Online]. Available: https://fastapi.tiangolo.com/

[12] Gemini Team et al., "Gemini: A family of highly capable multimodal models," Google DeepMind, Tech. Rep., 2023. [Online]. Available: https://arxiv.org/abs/2312.11805

[13] M. Ouhda, Z. Yousra, and B. Aksasse, "Smart harvesting decision system for date fruit based on fruit detection and maturity analysis using yolo and k-means segmentation," *Journal of Computer Science*, vol. 19, no. 10, pp. 1242–1252, 2023.

[14] A. Almutairi, J. Alharbi, S. Alharbi, H. Alhasson, S. Alharbi, and S. Habib, "Date fruit detection and classification based on its variety using deep learning technology," *IEEE Access*, vol. 12, pp. 104 042–104 053, 2024.

[15] S. Lipiński, S. Sadkowski, and P. Chwietczuk, "Application of ai in date fruit detection—performance analysis of yolo and faster r-cnn models," *Computation*, vol. 13, no. 6, p. 149, 2025.

[16] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2021.

# Appendix A

# Source Code Listings

# Appendix B

# Dockerfile and Environment Configuration

# Appendix C

# Sample Generative AI Quality Report

# Appendix D

# Dataset Sample Images