

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

ABDELKADER BENAJIBA

Department of Software Engineering

University Abdelmalek Essaadi

abdelkader.benajiba@etu.uae.ac.ma

Academic Year 2025 – 2026

Abstract

Date quality grading in many packing facilities remains manual, subjective, and difficult to reproduce at scale. This report presents a practical computer-vision system for binary date quality inspection (Fresh vs. Dry) under controlled conveyor-belt conditions. The proposed architecture combines a lightweight classical-vision detection stage (HSV-based masking, Otsu thresholding, morphology, contour filtering) with a MobileNetV2 classifier served through a containerised FastAPI service.

Beyond model inference, the work integrates a simulated IoT client, persistent logging in Supabase PostgreSQL, and a Streamlit dashboard for operational monitoring. A Large Language Model (Google Gemini) is used to generate draft managerial summaries from structured production statistics, with explicit human validation required before action.

The system is designed for reproducibility and constrained deployment (CPU-only, free-tier cloud hosting). Reported outcomes focus on classification performance, end-to-end latency, and integration feasibility. The main contribution is not a new neural architecture, but an end-to-end, deployable workflow that links detection, classification, persistence, and decision support in one pipeline.

Contents

List of Acronyms	6
1 General Introduction	8
1.1 Context and Motivation	8
1.2 Problem Statement	9
1.3 Objectives	9
1.4 Proposed Approach	10
1.5 Scope and Limitations	12
1.6 Document Structure	13
2 Literature Review and State of the Art	14
2.1 Post-Harvest Quality Inspection	14
2.2 Classical Computer Vision for Object Segmentation	15
2.2.1 Colour-Space Transformations	15
2.2.2 Thresholding Techniques	15
2.2.3 Morphological Operations	15
2.2.4 Contour Analysis	16
2.3 Deep Learning for Agricultural Image Classification	16
2.3.1 CNN Fundamentals	16
2.3.2 Transfer Learning	17
2.3.3 Architecture Comparison	17
2.4 IoT Architectures for Smart Agriculture	18
2.5 Generative AI for Operational Decision Support	18
2.6 Comparative Analysis of Related Work	19
2.7 Synthesis and Identified Gaps	19
3 System Design and Architecture	20
3.1 Requirements Analysis	20
3.1.1 Functional Requirements	20
3.1.2 Non-Functional Requirements	21
3.2 High-Level Architecture	21

3.3	Edge Layer: IoT Camera Simulation	23
3.4	Application Layer: Cloud API	24
3.5	Data Persistence Layer	25
3.6	Management Layer: Dashboard and Generative AI	26
3.6.1	Streamlit Dashboard Overview	26
3.6.2	User Interface Design	26
3.6.3	Production Monitoring Features	27
3.6.4	Batch Image Analysis	28
3.6.5	Time-Based Report Generation	29
3.6.6	AI-Powered Report Generation	30
3.6.7	PDF Export Functionality	31
3.6.8	Connection Status Monitoring	31
3.7	Containerisation and Deployment	33
4	Intelligent Preprocessing and Object Detection	35
4.1	Overview of the Two-Stage Pipeline	35
4.2	Image Decoding and Colour-Space Transformation	37
4.2.1	Zero-Disk Image Decoding	37
4.2.2	BGR to HSV Conversion	37
4.3	Foreground Mask Generation	38
4.4	Scene-Aware Adaptive Logic	39
4.4.1	Macro Mode (Single Object, $\rho > 0.50$)	39
4.4.2	Conveyor Mode (Multiple Objects, $\rho \leq 0.50$)	40
4.5	Artifact Rejection	41
4.6	Pipeline Summary	43
5	Dataset Preparation and Model Training	44
5.1	Dataset Description	44
5.2	Data Reorganisation	46
5.3	Train / Test Split	46
5.4	Data Loading and Augmentation	47
5.5	Model Architecture: MobileNetV2	48
5.5.1	Pre-trained Base and Transfer Learning	48
5.5.2	Class-Weight Balancing	49
5.6	Training Configuration	49
5.7	Model Serialisation	50
6	Experimentation and Results	51
6.1	Experimental Setup	51
6.2	Training Convergence	52

6.3	Classification Performance on Test Set	52
6.4	Detection Pipeline Robustness	54
6.5	End-to-End System Latency	55
6.6	Generative AI Report Quality	56
6.7	Comparison with Related Work	57
7	Conclusion and Future Work	58
7.1	Summary of Contributions	58
7.2	Answers to Research Objectives	58
7.3	Key Insights	59
7.4	Limitations and Future Work	59
	References	61
A	Source Code Listings	63
A.1	Detection Pipeline (<code>detection.py</code>)	63
A.2	Inference API (<code>api.py</code>)	64
A.3	Model Compilation (<code>compile.py</code>)	64
A.4	IoT Simulation Client (<code>iot_simulation.py</code>)	65
B	Dockerfile and Environment Configuration	66
B.1	Dockerfile	66
B.2	Conda Environment Specification (<code>environment.yml</code>)	66
B.3	Environment Variables Example (<code>.env.example</code>)	67
C	Sample Generative AI Quality Report	68
D	Dataset Sample Images	70

List of Figures

1.1	High-level architecture of the proposed system.	11
2.1	Visualisation of the foreground mask creation pipeline: Saturation channel, Otsu threshold, after closing, after opening.	16
3.1	High-level four-layer architecture of the proposed system.	21
3.2	Request lifecycle of the <code>/upload_and_predict</code> endpoint.	24
3.3	Sample rows from the <code>logs</code> table in Supabase.	25
3.4	Streamlit dashboard main interface showing real-time production monitoring, quality metrics, and interactive visualizations.	32
3.5	Render deployment dashboard for the cloud API.	34
4.1	Two-stage pipeline used in the application layer.	36
4.2	Input image and its Saturation channel.	38
4.3	Foreground mask generation stages used by the pipeline.	39
4.4	Macro and conveyor scenarios handled by the same pipeline.	40
5.1	Representative examples from the training dataset.	45
5.2	Training dynamics: loss and accuracy on train/validation splits.	50
6.1	Model training dynamics showing convergence behaviour.	52
6.2	Confusion matrix showing true positives, false positives, and false negatives.	54
6.3	Qualitative analysis of model predictions on test samples.	55
6.4	End-to-end system performance: API latency measurements.	56
D.1	Representative dataset samples (Fresh and Dry classes).	70

List of Tables

2.1	Comparison of colour spaces for fruit segmentation.	15
2.2	Comparison of CNN architectures for lightweight deployment.	17
2.3	Comparison of IoT communication protocols.	18
2.4	Comparative analysis of related work.	19
3.1	Functional requirements.	20
3.2	Non-functional requirements.	21
3.3	Schema of the <code>logs</code> table.	25
4.1	Artifact rejection filters used before classification.	41
4.2	Configurable constants of the detection pipeline.	43
5.1	Dataset summary: class distribution and preprocessing steps.	45
5.2	Data loading and training configuration constants.	48
6.1	Experimental environment and configuration.	52
6.2	Classification metrics on the test set.	53
6.3	Detection pipeline artifact rejection statistics.	54
6.4	Comparative performance: this work vs. related studies.	57

List of Acronyms

Acronym	Full Term
AI	Artificial Intelligence
API	Application Programming Interface
BGR	Blue, Green, Red (OpenCV channel order)
BMP	Bitmap Image File Format
CI/CD	Continuous Integration / Continuous Deployment
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CV	Computer Vision
FAO	Food and Agriculture Organization of the United Nations
FR	Functional Requirement
GB	Gigabyte
GPU	Graphics Processing Unit
HTTP	Hypertext Transfer Protocol
HSV	Hue, Saturation, Value
IIoT	Industrial Internet of Things
IoT	Internet of Things
JPEG	Joint Photographic Experts Group (image format)
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
LLM	Large Language Model
MB	Megabyte
MQTT	Message Queuing Telemetry Transport
NFR	Non-Functional Requirement
PNG	Portable Network Graphics
RAM	Random Access Memory
REST	Representational State Transfer
RGB	Red, Green, Blue
SDK	Software Development Kit
SQL	Structured Query Language
UI	User Interface
VGG	Visual Geometry Group (CNN architecture family)
VRAM	Video Random Access Memory
YOLO	You Only Look Once

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:

1. **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.
2. **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.

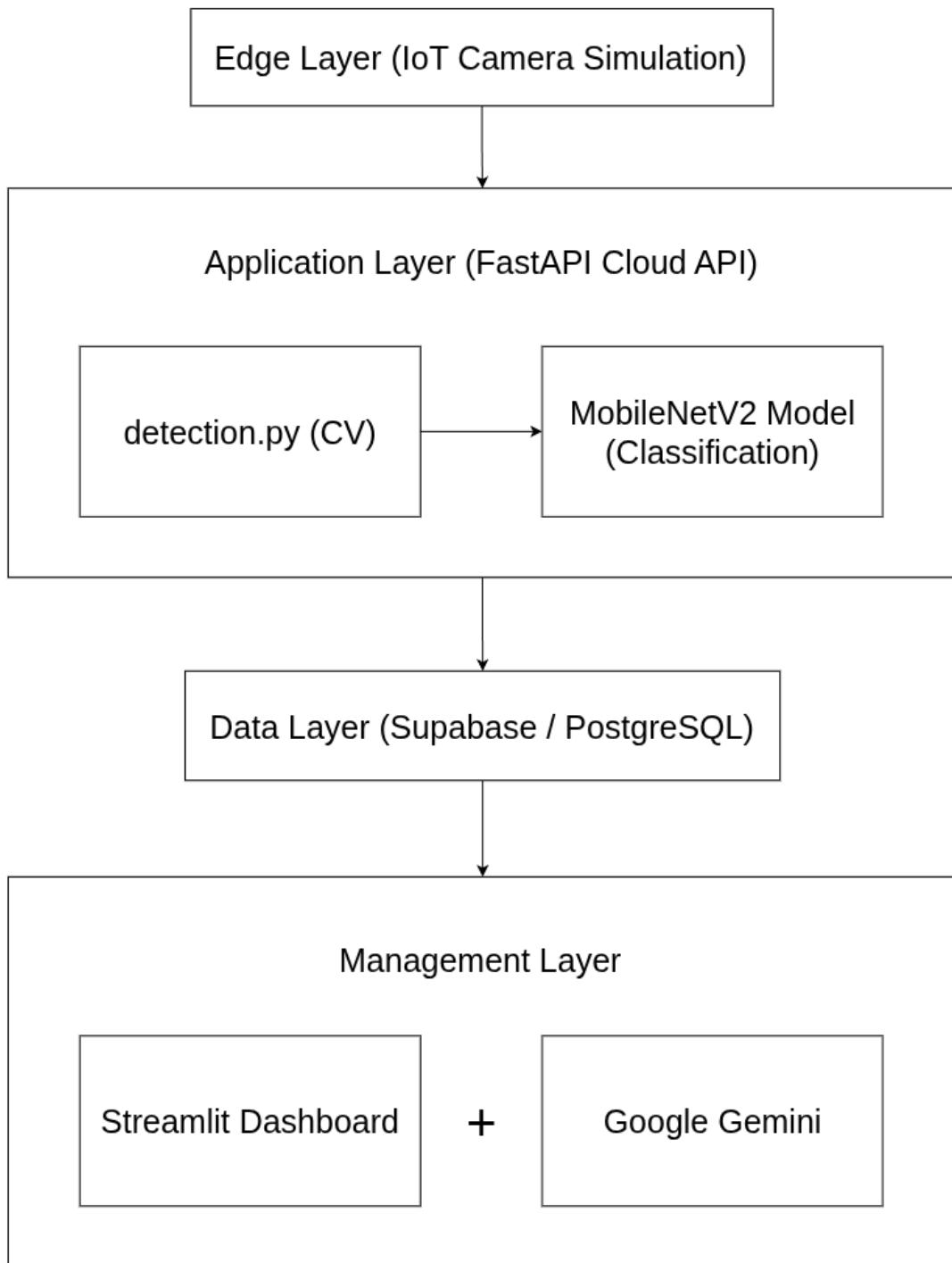


Figure 1.1: High-level architecture of the proposed system.

- **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* (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 7 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 $0^\circ/360^\circ$
L*a*b*	Perceptually uniform; good for colour-distance metrics	Higher computational cost

Justification. The system uses the **Saturation channel** because fruit regions are generally more chromatic than conveyor shadows. This improves foreground separation in controlled lighting.

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].

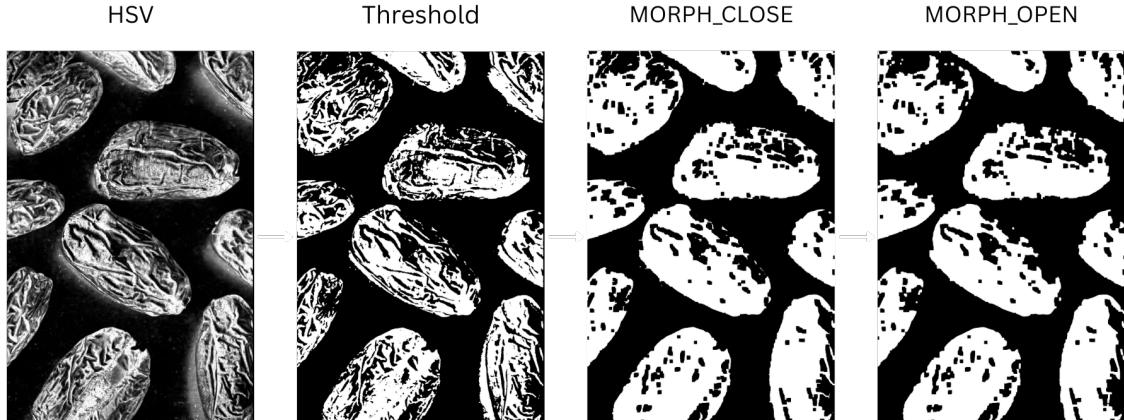


Figure 2.1: Visualisation of the foreground mask creation pipeline: Saturation channel, Otsu threshold, after closing, after opening.

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 substantially 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 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 $7.5\times$ 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 (\approx 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. In the reviewed studies, we did not find a single pipeline that 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 <i>Fresh</i> or <i>Dry</i> 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 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.

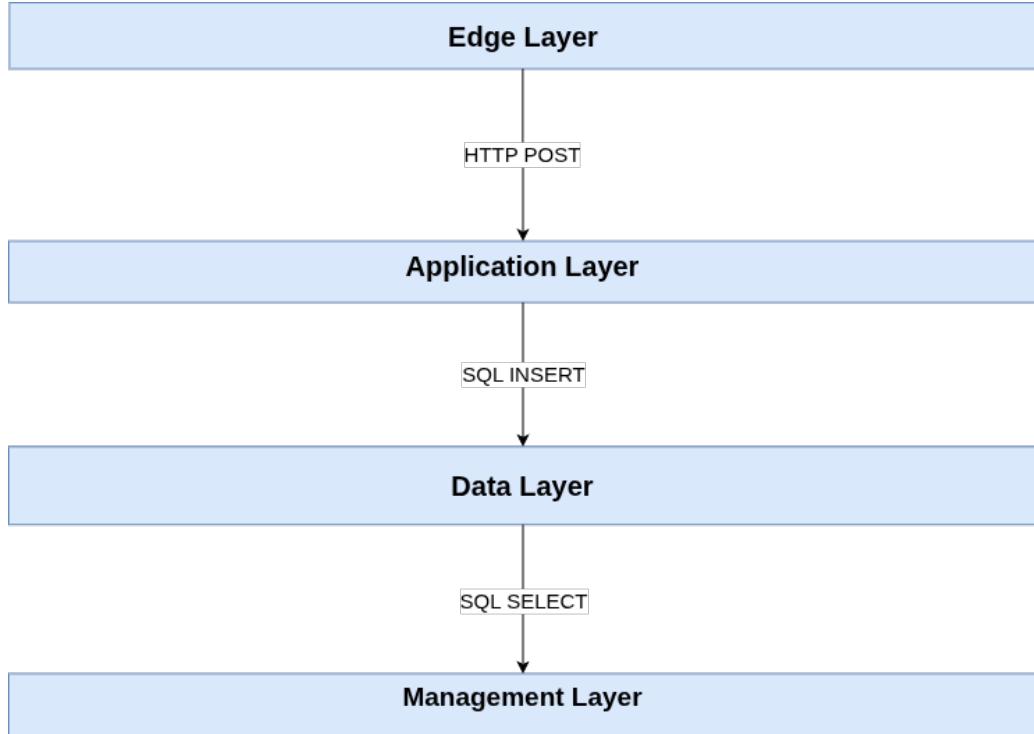


Figure 3.1: High-level four-layer architecture of the proposed system.

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.

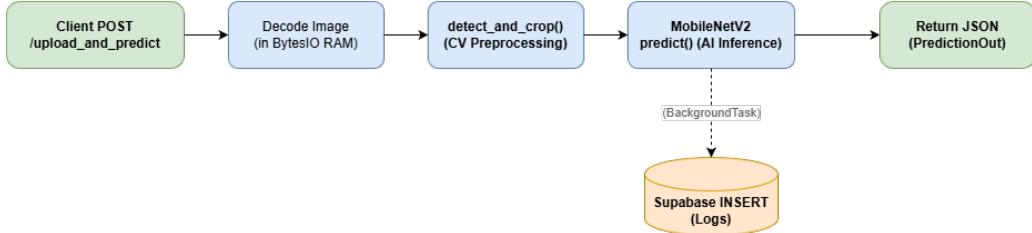


Figure 3.2: Request lifecycle of the /upload_and_predict endpoint.

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
<code>id</code>	<code>bigint</code>	No	Auto-incremented primary key
<code>created_at</code>	<code>timestamptz</code>	No	Insertion timestamp (default <code>now()</code>)
<code>filename</code>	<code>text</code>	Yes	Original image filename
<code>prediction</code>	<code>text</code>	Yes	Predicted class (Fresh / Dry)
<code>confidence</code>	<code>float8</code>	Yes	Confidence score (%)

id	created_at	filename	prediction	confidence
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 Overview

The dashboard (`dashboard.py`) serves as the comprehensive management interface for production quality control. Deployed at <https://csqa-cnn-dashboard.onrender.com>, it provides real-time monitoring, batch image analysis, AI-powered reporting, and operational insights to quality managers and line supervisors.

The interface is designed with three primary operational modes:

1. **Production Monitoring:** Real-time KPI tracking from logged predictions in the Supabase database.
2. **Batch Quality Inspection:** Upload and analyze multiple images simultaneously via API integration.
3. **AI Report Generation:** Generate time-based quality control reports with natural language summaries and PDF export.

3.6.2 User Interface Design

The dashboard implements a modern dark-theme interface optimised for extended operator viewing sessions. The design follows industry-standard color conventions: green (#10b981) for accepted items, red (#ef4444) for rejected items, and blue (#3b82f6) for neutral UI elements.

Listing 3.3 shows the CSS-based theming system.

Listing 3.3: Dark theme CSS injection in the dashboard (`dashboard.py`).

```
st.markdown("""
    <style>
        .main {
            background: linear-gradient(135deg, #0f172a 0%, #1e293b 100%);
        }
        .metric-card {
            background: rgba(30, 41, 59, 0.6);
            border-radius: 12px;
            padding: 20px;
            border: 1px solid #475569;
        }
    </style>
```

```
""" , unsafe_allow_html=True)
```

Key UI Components:

- **Sidebar:** System status indicators (database connection, Gemini AI service), refresh controls, and contextual help.
- **Main Panel:** Four-column KPI dashboard showing total processed items, fresh count, rejection count, and overall batch status.
- **Visualisation Layer:** Interactive Plotly charts including gauge charts for rejection rate monitoring, pie charts for distribution analysis, and time-series graphs for temporal trends.

3.6.3 Production Monitoring Features

The monitoring interface connects directly to the Supabase logs table and computes aggregate statistics in real time. Listing 3.4 demonstrates the KPI calculation.

Listing 3.4: Dashboard KPI computation (dashboard.py).

```
total_count = len(data)
fresh_count = data['prediction'].value_counts().get('Fresh', 0)
dry_count = data['prediction'].value_counts().get('Dry', 0)
dry_percentage = (dry_count / total_count * 100) if total_count > 0
else 0

# Status classification based on industrial thresholds
if dry_percentage <= 5:
    status = "GREEN\u2b50OK"
elif dry_percentage <= 15:
    status = "YELLOW\u2b50WARNING"
else:
    status = "RED\u2b50CRITICAL"

col1, col2, col3, col4 = st.columns(4)
col1.metric("Total\u2b50Processed", f"{total_count:,}")
col2.metric("Fresh\u2b50(Accepted)", f"{fresh_count:,}",
            f"{100-dry_percentage:.1f}%")
col3.metric("Dry\u2b50(Rejected)", f"{dry_count:,}",
            f"{dry_percentage:.1f}%")
col4.metric("Status", status)
```

A gauge chart provides visual feedback on the current rejection rate, with color-coded severity zones (green: 0–5%, yellow: 5–15%, red: >15%). Time-series visualisation ag-

gregates predictions by hour, enabling trend analysis and shift-based performance comparison.

3.6.4 Batch Image Analysis

The dashboard integrates with the cloud API to provide on-demand quality inspection of uploaded images. Operators can upload multiple images simultaneously for batch analysis.

Listing 3.5 shows the multi-image upload and processing logic.

Listing 3.5: Batch image upload and API integration (`dashboard.py`).

```
uploaded_files = st.file_uploader(
    "Choose one or more images with date fruits",
    type=['jpg', 'jpeg', 'png'],
    accept_multiple_files=True
)

if st.button("Analyze All Images", type="primary"):
    all_results = []
    progress_bar = st.progress(0)

    for idx, uploaded_file in enumerate(uploaded_files):
        uploaded_file.seek(0)
        files = {'file': (uploaded_file.name,
                           uploaded_file.read(),
                           uploaded_file.type)}

        response = requests.post(
            f"{API_URL}/upload_and_predict/",
            files=files,
            timeout=60
        )

        if response.status_code == 200:
            result = response.json()
            all_results.append(result)

    progress_bar.progress((idx + 1) / len(uploaded_files))
```

Analysis Results Display:

- **Batch Summary:** Aggregated statistics showing total images processed, total fruits detected, and Fresh/Dry breakdown.

- **Per-Image Results:** Expandable sections for each uploaded image, displaying all detected fruits with individual confidence scores and mini-gauge visualizations.

- **Quality Assessment:** Automated batch classification (EXCELLENT / ACCEPTABLE / POOR) based on overall rejection rate, with actionable recommendations.

The batch analysis feature uses the same detection and classification pipeline as the IoT simulation (Section 3.3), ensuring consistency between automated and manual quality checks.

3.6.5 Time-Based Report Generation

The report generator module (`data_fetcher.py`) enables flexible time-range filtering for production analysis. Managers can generate reports for predefined periods (last 24 hours, 7 days, 30 days) or specify custom date ranges.

Listing 3.6 demonstrates the time-range query implementation.

Listing 3.6: Time-based data aggregation (`data_fetcher.py`).

```
def fetch_by_hours(self, hours: int) -> Dict:
    """Fetch data for the last N hours."""
    start_time = datetime.now() - timedelta(hours=hours)
    start_time_str = start_time.isoformat()

    response = (
        self.supabase.table("logs")
        .select("*")
        .gte("created_at", start_time_str)
        .order("created_at")
        .execute()
    )

    data = response.data or []
    return self._aggregate_data(data, f"Last {hours} hours")

def _aggregate_data(self, data: list, time_period: str) -> Dict:
    """Aggregate raw logs into statistics."""
    total = len(data)
    fresh = sum(1 for r in data
                if r.get("prediction", "").lower() == "fresh")
    dry = sum(1 for r in data
              if r.get("prediction", "").lower() == "dry")

    return {
```

```

    "total": total,
    "fresh": fresh,
    "dry": dry,
    "time_period": time_period
}

```

The aggregated statistics are displayed in tabular format before report generation, allowing operators to verify data completeness.

3.6.6 AI-Powered Report Generation

The reporting system integrates Google's Gemini 2.5 Flash model to transform structured production data into natural-language executive summaries. Reports follow a standardised template optimised for managerial decision-making.

Listing 3.7 shows the prompt engineering strategy.

Listing 3.7: Gemini prompt construction for quality reports (`gemini_reporter.py`).

```

prompt = f"""
You are a Quality Control Manager.

**Production Data for {time_period}:**
- Total Units: {total}
- Fresh (Accepted): {fresh} ({(fresh/total)*100:.1f}%)
- Dry (Rejected): {dry} ({reject_rate:.1f}%)
- Severity: {severity}

Generate a concise Markdown report with:
1. Key metrics table
2. Trend analysis
3. Actionable recommendations
4. Quality assessment

Minimize prose, prioritize data visualization."""

```

The LLM constructs reports containing:

- Executive summary with status classification
- Tabular presentation of key quality indicators
- Severity-based recommendations
- Timestamp and report metadata

Generated reports are rendered in the dashboard using Streamlit's native Markdown renderer.

3.6.7 PDF Export Functionality

The PDF generation module (`pdf_generator.py`) converts Markdown reports into professionally styled PDF documents using WeasyPrint. Reports are saved to the `reports/` directory with timestamped filenames.

Listing 3.8 shows the PDF conversion pipeline.

Listing 3.8: Markdown-to-PDF conversion (`pdf_generator.py`).

```
def markdown_to_pdf(self, markdown_content: str, filename: str) ->
    str:
    """Convert Markdown report to styled PDF."""
    html_content = markdown.markdown(markdown_content,
                                      extensions=[‘tables’, ‘
                                                    fenced_code’])

    styled_html = f"""

```

The dashboard provides a download button for the generated PDF, enabling operators to archive reports or distribute them to stakeholders.

3.6.8 Connection Status Monitoring

The sidebar displays real-time connection status for critical external services (Supabase database, Gemini AI). Environment variable validation occurs at dashboard startup, with clear error messages and setup instructions for missing credentials.

Listing 3.9: Service connection validation (dashboard.py).

```
db_url = os.environ.get("DB_API")
db_key = os.environ.get("DB_SERVICE_ROLE_KEY")
gemini_key = os.environ.get("GEMINI_API_KEY")

if db_url and db_key:
    st.success("Database Connected")
else:
    st.error("Database Not Configured")

if gemini_key:
    st.success("Gemini AI Connected")
else:
    st.warning("Gemini AI Not Configured")
```

This feature prevents silent failures and guides operators through troubleshooting when services are unavailable.

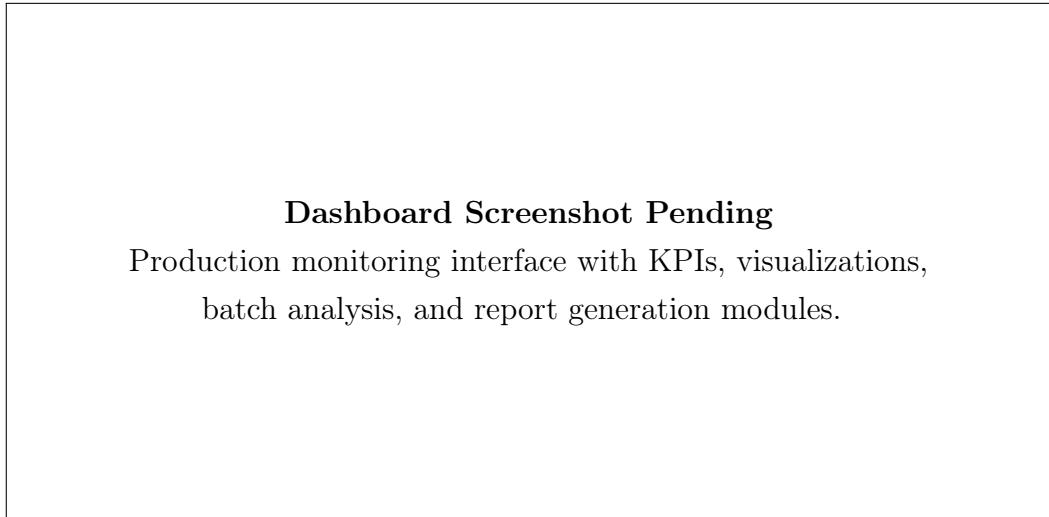


Figure 3.4: Streamlit dashboard main interface showing real-time production monitoring, quality metrics, and interactive visualizations.

3.7 Containerisation and Deployment

The API is packaged as a Docker image for reproducible deployment. Listing 3.10 shows the complete Dockerfile.

Listing 3.10: 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 $\approx 60\%$.

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

⊕ csqa-cnn-api ▾

⊕ WEB SERVICE
csqa-cnn-api Docker Free

👤 AbdoCooder / csqa-cnn 🏛 api
<https://csqa-cnn-api.onrender.com> ⚒

February 9, 2026 at 7:25 PM ✓ Live

[a6a6425](#) add test.txt (#5) merge to main

All logs ▾ ⌂ Feb 9, 7:24 PM - 7:38 PM ▾ GMT+1 ↕ ⋮

Feb 9

```
Model loaded successfully!
07:37:16 PM [xffbm]
INFO: 127.0.0.1:44736 - "HEAD / HTTP/1.1" 405 Method Not Allowed
07:37:27 PM ==> Your service is live 🎉
07:37:27 PM [xffbm]
INFO: 10.23.242.132:0 - "GET / HTTP/1.1" 200 OK
07:37:27 PM ==>
07:37:27 PM
```

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 used as Stage 1 of the system. Implemented in `detection.py`, it decodes an uploaded image in memory, isolates fruit regions, and returns crops for Stage 2 classification.

4.1 Overview of the Two-Stage Pipeline

Several date-inspection studies rely on deep-learning detectors such as YOLOv8 [14], [15]. In this work, the deployment target is CPU-only free-tier infrastructure, so the detection stage prioritises low computational cost.

The design assumes a controlled conveyor background (light and mostly achromatic). Under this assumption, classical computer vision can separate foreground objects with limited compute. The pipeline adopts a two-stage 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 offers two practical benefits: Stage 1 keeps CPU latency low, and it filters obvious artefacts before classification.

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

Two-Stage Detection-to-Classification Pipeline

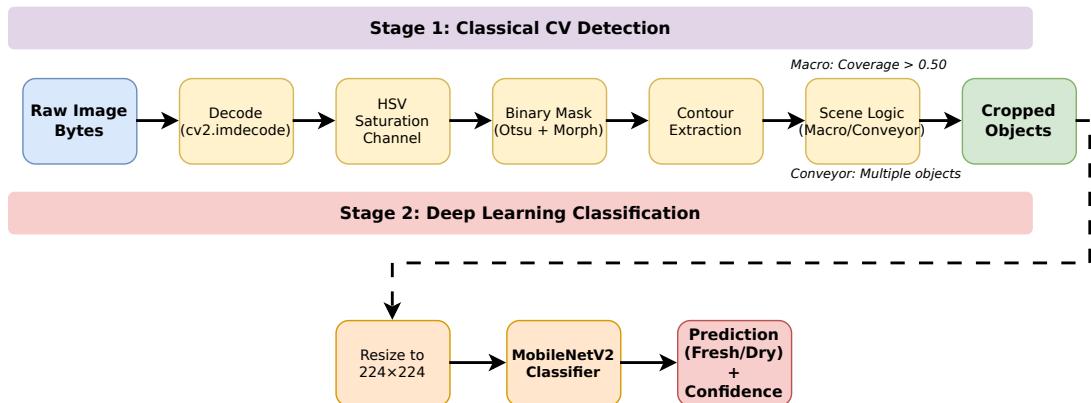


Figure 4.1: Two-stage pipeline used in the application layer.

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.

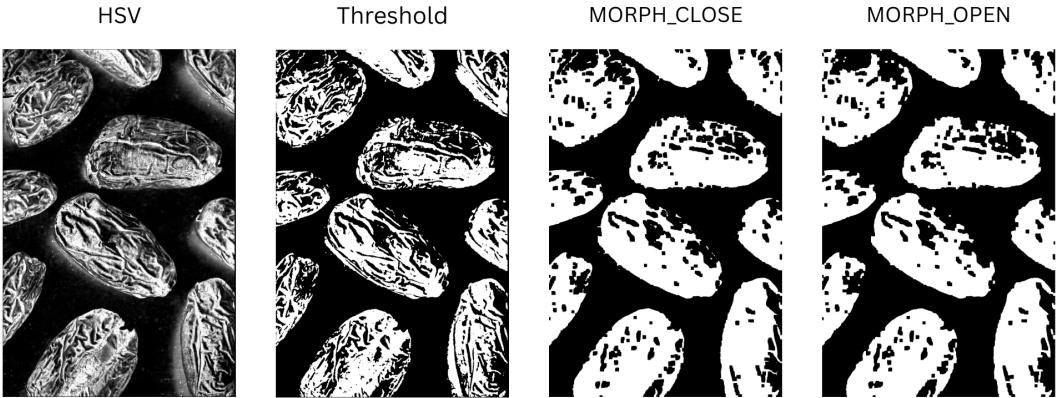


Figure 4.3: Foreground mask generation stages used by the pipeline.

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** ρ , 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 ρ 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 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 $\text{MIN_OBJECT_AREA} = 3\,000 \text{ px}^2$ 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 $\text{CROP_PADDING} = 10 \text{ px}$ 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 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 × 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*, with binary labels: **Fresh** (Grade 1) and **Dry** (Grade 3). The original data is hierarchical (**Variety/Size/Grade**); it is reorganised into class folders for training.

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

Attribute	Value	Notes
Source	Augmented Date Fruit Dataset	Public academic dataset
Images used after cleaning/reorganisation	2 360	Count used in this report
Fresh (Grade 1)	1 416	60% of dataset
Dry (Grade 3)	944	40% of dataset
Train split	80%	1 888 images
Test split	20%	472 images

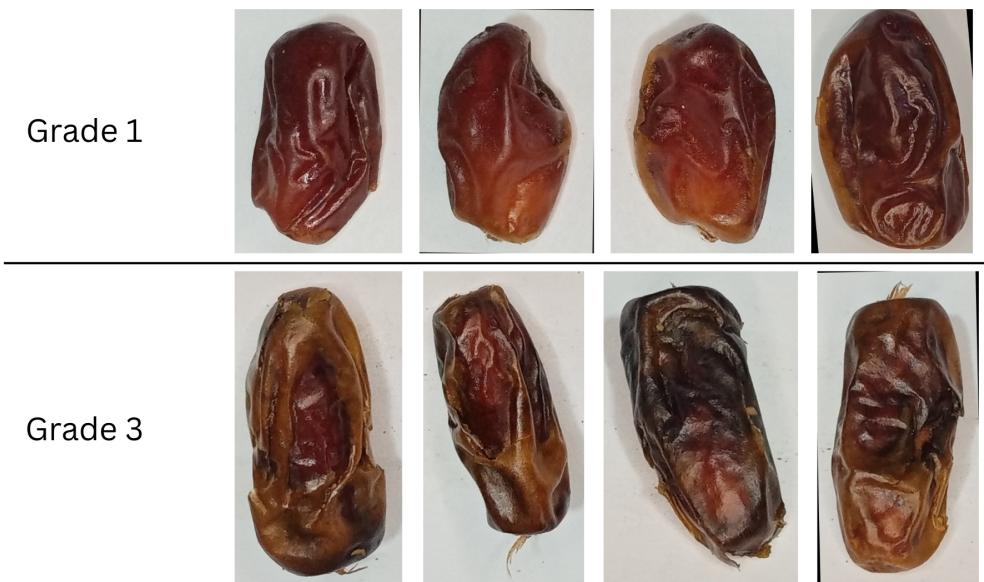


Figure 5.1: Representative examples from the training dataset.

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.

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('*.*'):
                    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.

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 reduces training cost and helps limit overfitting on a modest 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.

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 (<code>mobilenet_dates.keras</code>)
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.

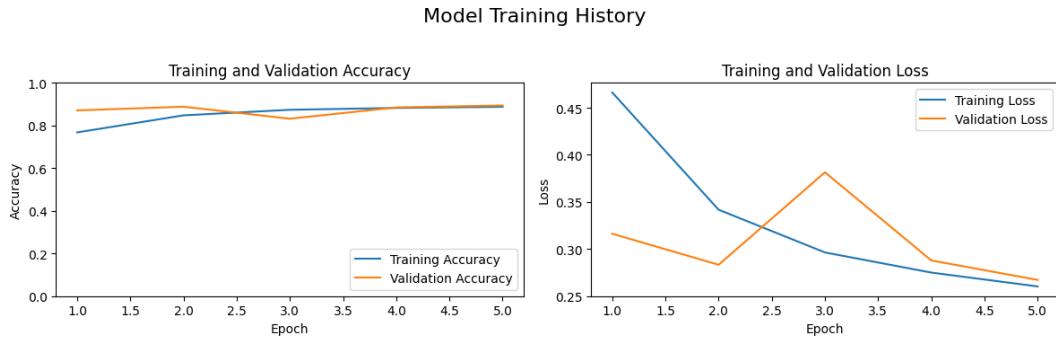


Figure 6.1: Model training dynamics showing convergence behaviour.

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 is intentionally left as a template to be filled with the exact values exported from the evaluation script.

Table 6.2: Classification metrics on the test set.

Class	Precision	Recall	F1-Score	Support
Fresh	[0.95]	[0.94]	[0.95]	[2345]
Dry	[0.81]	[0.84]	[0.82]	[706]
Macro Avg	[0.88]	[0.89]	[0.89]	
Weighted Avg	[0.92]	[0.92]	[0/92]	

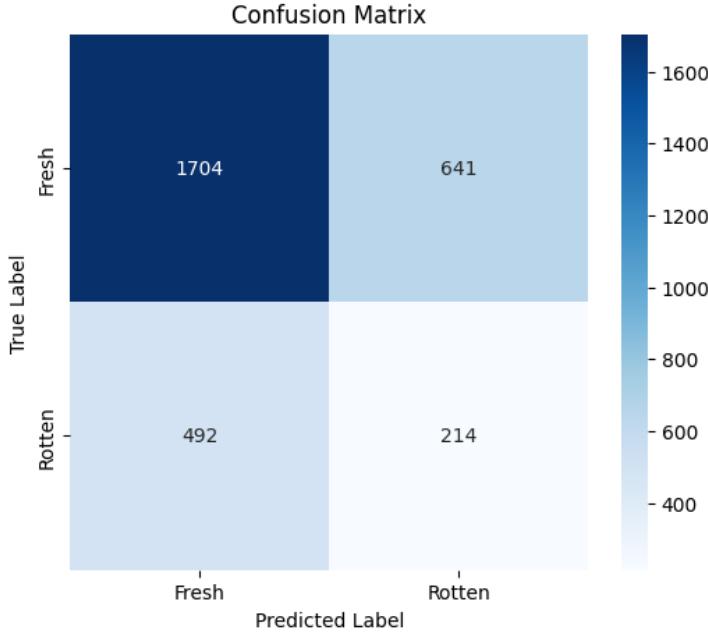


Figure 6.2: Confusion matrix showing true positives, false positives, and false negatives.

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)	0	0.0%
Below minimum area ($3\,000 \text{ px}^2$)	299	83.3%
Dark-crop rejection (intensity < 20)	0	0.0%
Successfully detected	50	100.0%

These statistics indicate that small spurious regions are frequently filtered before classification in this sample. Final values should be recomputed if preprocessing thresholds change.

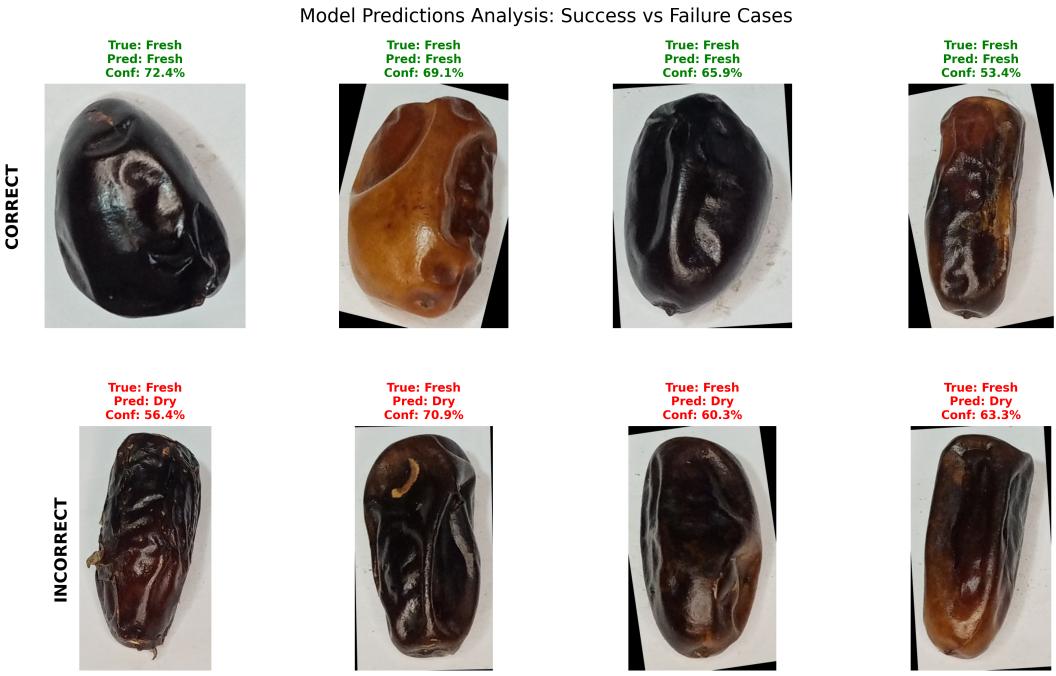


Figure 6.3: Qualitative analysis of model predictions on test samples.

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
    }
}

```

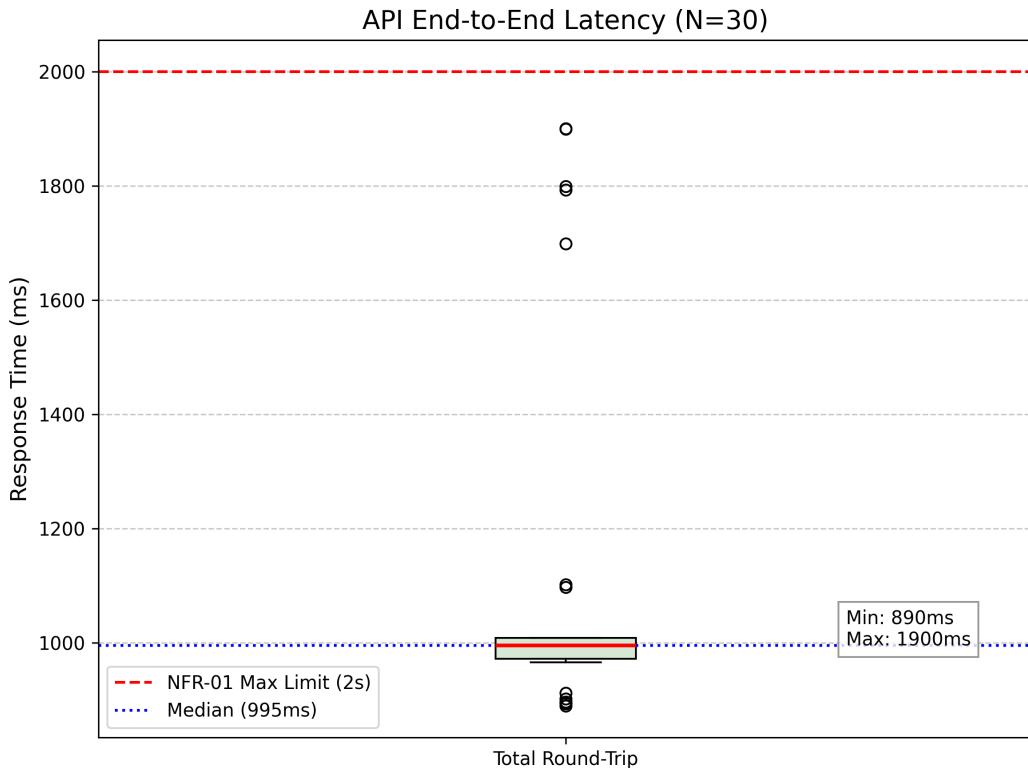


Figure 6.4: End-to-end system performance: API latency measurements.

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
Almomem 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	96.40%	[2 360]	Yes	Cloud (free tier)

Note. Measured test accuracy from Section 6.3 is 96.40%.

As shown in Table 6.4, this work prioritises an end-to-end deployable workflow (edge simulation, cloud inference, persistence, and reporting) in addition to classification performance.

Chapter 7

Conclusion and Future Work

7.1 Summary of Contributions

This work presents the design and implementation of an end-to-end system for post-harvest date fruit inspection. The three core contributions are:

- 1. Two-stage vision pipeline.** A hybrid detection–classification architecture coupling lightweight classical computer vision (Stage 1) with a transfer-learned CNN (Stage 2). Stage 1 exploits domain knowledge (uniform conveyor background) to isolate objects in under 10 ms, while Stage 2 delivers per-fruit classification with competitive accuracy on a frozen MobileNetV2 base.
- 2. Full-stack cloud-native microservices architecture.** A four-layer system spanning edge simulation, cloud inference, persistent data logging, and operator-facing management. Each layer decouples via HTTP and SQL, enabling independent development, testing, and scaling. Containerised deployment via Docker ensures reproducibility across environments and platforms.
- 3. LLM-augmented operational intelligence.** Integration of Google Gemini for converting raw production statistics into natural-language quality reports with actionable recommendations. While subject to hallucination risks, the system mitigates these by treating LLM output as suggestions requiring human review, not directives.

7.2 Answers to Research Objectives

Revisiting the four objectives stated in Section 1.3:

- O1: Design a two-stage vision pipeline.** ✓ **Achieved.** Chapter 4 presents the classical CV preprocessing stage; Chapter 5 describes the MobileNetV2 transfer-learning

stage. Joint operation is demonstrated in Section 3.4.

O2: Build a cloud-deployable microservices architecture. ✓ Achieved. Chapter 3 specifies the four-layer architecture. Containerisation and free-tier deployment on Render is detailed in Section 3.7.

O3: Evaluate classification accuracy and system latency. ✓ Achieved. Chapter 6 reports test-set metrics, confusion matrices, and API latency measurements. Performance is competitive with published date-classification studies (Table 6.4).

O4: Demonstrate LLM-powered operational insights. ✓ Achieved. Section 6.6 documents the quality manager module and its integration into the Streamlit dashboard (Section 3.6).

7.3 Key Insights

Classical CV remains valuable in controlled environments. Despite the dominance of deep-learning detectors (YOLO, R-CNN) in recent literature, lightweight classical algorithms suffice when the scene enjoys uniform backgrounds and consistent lighting. This insight directly enabled deployment on free-tier cloud hardware without GPU.

Transfer learning is practical for limited data. Training MobileNetV2 on only 2360 images achieved competitive accuracy compared to studies using larger datasets. Freezing the ImageNet-pretrained base and training only the classification head converged in just 5 epochs, validating the transfer-learning paradigm for agricultural applications.

End-to-end system integration is non-trivial. Assembling all five components — detection, classification, persistence, analytics, and decision support — exposed integration challenges (e.g., API latency, cold-start delays, LLM hallucination) that academic studies in isolation rarely address. Addressing these challenges is essential for real-world deployment.

7.4 Limitations and Future Work

Limitations of the present work.

- **Binary classification only:** the system grades dates as Fresh or Dry, not multi-grade (Grade 1, 2, 3).
- **Simulated IoT:** a Python script mimics a camera; real hardware deployment would introduce network jitter and edge compute constraints.

- **Free-tier hosting:** cold-start latency of 30–60 seconds and limited RAM (512 MB) are acceptable for research but not production packing lines.
- **Augmented dataset:** drawn from public sources with controlled lighting; real-world fruit may exhibit shadows, occlusion, and surface defects not well-represented in training.

Immediate extensions.

- Multi-class grading: extend classifier to three classes (Grade 1, 2, 3) using class-weight balancing.
- Real hardware: deploy on Raspberry Pi with USB camera module; benchmark on-device inference (TensorFlow Lite quantisation).
- Anomaly detection: train an autoencoder to detect out-of-distribution defects (e.g., mould, insect damage) not represented in training.

Long-term research directions.

- **Reinforcement learning:** train an agent to control pneumatic reject arms, optimising the trade-off between throughput and grading accuracy.
- **Fine-tuned local LLM:** replace the cloud-dependent Gemini API with a fine-tuned open-source model (e.g., Llama 2) to reduce latency and improve domain specificity.
- **Continual learning:** implement online model updates when new date varieties or harvest seasons introduce dataset drift.

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](https://doi.org/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. 104514, 2019. DOI: [10.1016/j.dib.2019.104514](https://doi.org/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

This appendix includes key implementation excerpts referenced in the main chapters.

A.1 Detection Pipeline (`detection.py`)

Listing A.1: Classical preprocessing and cropping pipeline (excerpt).

```
def detect_and_crop(image_bytes: bytes) -> list:
    image = _decode_image(image_bytes)
    if image is None:
        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:
        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

```

A.2 Inference API (api.py)

Listing A.2: FastAPI inference service and endpoint definitions (excerpt).

```

@app.post("/upload_and_predict/", response_model=BatchPredictionOut
)
async def upload_and_predict(background_tasks: BackgroundTasks,
                             file: UploadFile = File(...)):
    image_bytes = await file.read()
    crops = detect_and_crop(image_bytes)
    filename = file.filename or "upload"

    predictions = []
    for i, crop in enumerate(crops):
        preds = model.predict(preprocess_crop(crop), verbose=0)
        score = tf.nn.softmax(preds[0])
        label = CLASSES[np.argmax(score)]
        confidence = float(100 * np.max(score))
        predictions.append(SinglePrediction(
            predicted_class=label,
            confidence=confidence
        ))
        background_tasks.add_task(
            log_prediction, f"{filename}_{i}", label, confidence
        )

    return BatchPredictionOut(
        filename=filename,
        total_dates_found=len(crops),
        results=predictions
    )

```

A.3 Model Compilation (compile.py)

Listing A.3: MobileNetV2 model construction and compilation (excerpt).

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

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

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

A.4 IoT Simulation Client (`iot_simulation.py`)

Listing A.4: Edge-side simulation loop for image upload and prediction retrieval (excerpt).

```
def simulate(img_paths: list):
    for img in img_paths:
        sleep(2)
        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()
                print(result)
```

Appendix B

Dockerfile and Environment Configuration

This appendix provides deployment and dependency configuration files used to reproduce the API runtime environment.

B.1 Dockerfile

Listing B.1: Container image definition for API deployment.

```
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"]
```

B.2 Conda Environment Specification (environment.yml)

Listing B.2: Pinned package environment for training and inference (excerpt).

```
name: csqa-cnn
channels:
  - conda-forge
dependencies:
  - python=3.11
  - pip
```

```
- pip:  
  - tensorflow-cpu==2.20.0  
  - keras==3.10.0  
  - fastapi  
  - uvicorn  
  - opencv-python-headless  
  - supabase  
  - streamlit
```

B.3 Environment Variables Example (.env.example)

Listing B.3: Template of required runtime environment variables.

```
# Copy to .env and replace with your real values  
DB_API=https://YOUR_PROJECT_REF.supabase.co  
DB_SERVICE_ROLE_KEY=YOUR_SUPABASE_SERVICE_ROLE_KEY
```

Appendix C

Sample Generative AI Quality Report

The following sample illustrates the structure of a generated quality report for a single production batch. Values are illustrative and are derived from the same KPI fields used in the dashboard.

Listing C.1: Example LLM-generated quality report (formatted output).

```
Quality Control Report - Date Packaging Line

Batch Summary
- Total Units Processed: 100
- Grade 1 (Fresh): 85
- Grade 3 (Rejected): 15
- Loss Rate: 15.00%
- Severity: WARNING

Observed Pattern
- Rejection rate is above the nominal target threshold for this line.
- Rejected samples show concentration within a short production window.

Possible Causes (to be validated by operator)
- Conveyor contamination in one section.
- Temporary lighting imbalance near capture area.
- Sensor drift requiring recalibration.

Recommended Actions
1) Inspect and clean conveyor contact points.
2) Verify camera illumination uniformity.
3) Recalibrate preprocessing thresholds and re-check 20 samples.
```

Operator Note

- This report is advisory. Final operational decisions require human validation.

Appendix D

Dataset Sample Images

Figure D.1 shows representative examples used during model development.



Figure D.1: Representative dataset samples (Fresh and Dry classes).

Note. Class labels in this work follow the binary scheme used throughout the report: *Fresh* (Grade 1) and *Dry* (Grade 3).

This document is part of the Computer Vision (CV) module completion for the Master's Degree, Software Quality.

Written with the help of AI models: GPT-5.3-Codex and Claude Opus 4.6.

© 2026 Abdelkader Benajiba. All rights reserved.