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.** Human inspectors remain the primary quality gate in most date packing facilities. Their assessments are influenced by lighting, fatigue, and individual perception, leading to inconsistent grading across shifts and facilities.
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.

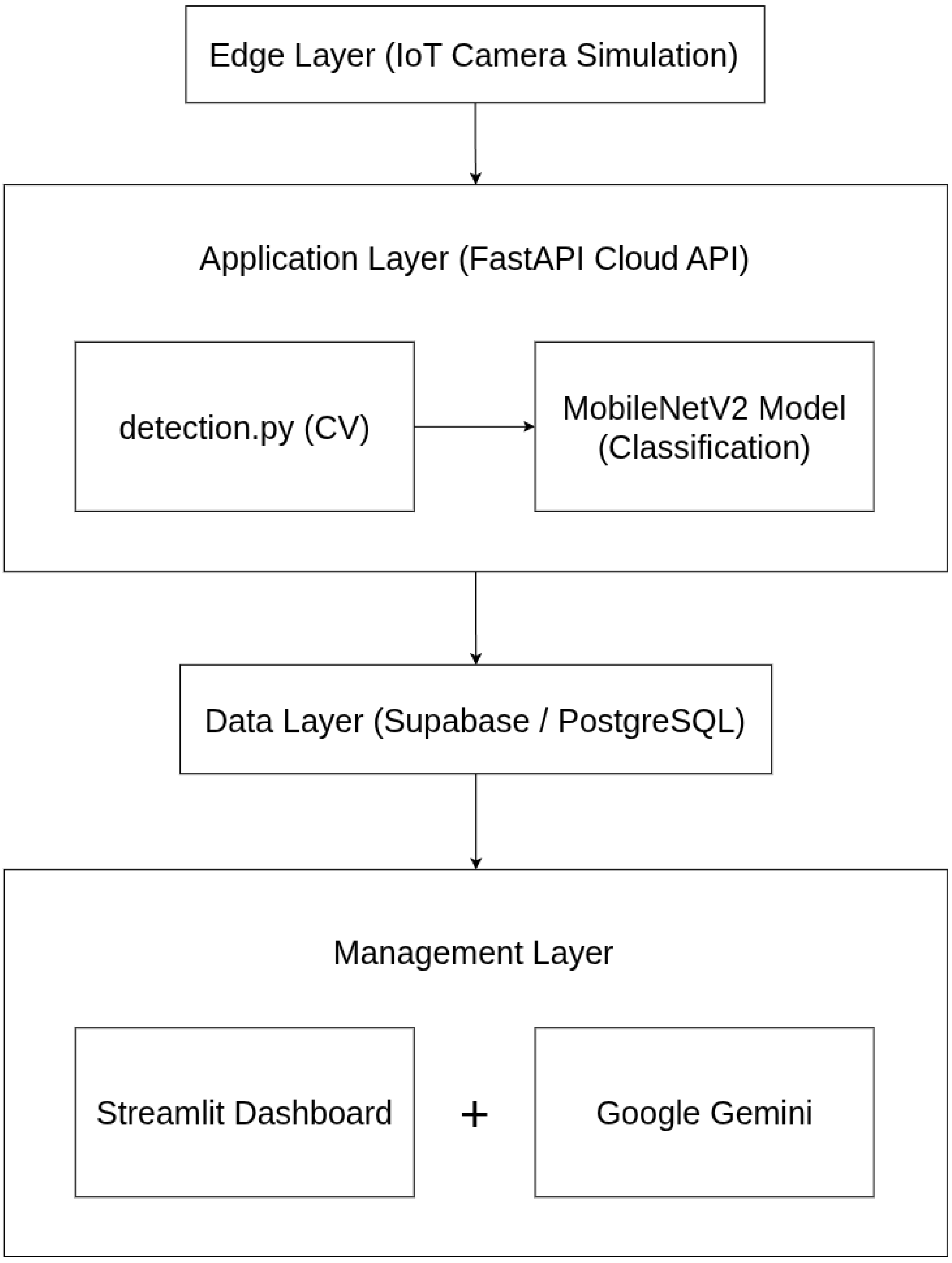
O3. Simulate an IoT edge device that mimics a conveyor-mounted industrial camera, continuously capturing and transmitting images to the cloud service with built-in network resilience.

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)
Application Layer (FastAPI Cloud API)
detection.py (CV)
MobileNetV2 Model (Classification)
Data Layer (Supabase / PostgreSQL)
Management Layer
Streamlit Dashboard
+
Google Gemini

- **Edge Layer.** A Python script (`iot_simulation.py`) simulates a camera mounted above a conveyor belt. It iterates over a folder of test images, transmitting each one to the cloud API via HTTP POST with a configurable delay to mimic the belt speed.

- **Application Layer.** A FastAPI application (`api.py`), containerised with Docker and hosted on Render, receives the uploaded image, decodes it in memory (zero-disk architecture), passes it through the two-stage pipeline (detection → classification), and returns a JSON response containing the predicted class and confidence score.

- **Data Persistence Layer.** Every prediction is asynchronously logged to a Supabase PostgreSQL database, creating a historical record suitable for time-series analysis without blocking the inference response.

- **Management Layer.** A Streamlit dashboard queries the database, visualises production metrics (rejection rate, class distribution over time), and can invoke Google Gemini to generate natural-language quality reports acting as a virtual Quality Manager.

The two-stage vision pipeline — which is the core technical contribution — first applies classical computer vision (HSV conversion, Gaussian blur, Otsu's adaptive thresholding, and morphological closing/opening) to isolate individual date fruits from the background, and then feeds each cropped region into a MobileNetV2 CNN pre-trained on ImageNet and fine-tuned on date fruit images for binary classification (*Fresh* vs. *Dry*).

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/Rotten* (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 ?? details the classical computer vision preprocessing pipeline used for object detection and isolation.

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

Chapter ?? 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

Date fruit quality is traditionally assessed by human graders who evaluate colour, texture, size, and the presence of surface defects [2]. While effective for small batches, manual inspection introduces three well-documented problems: (i) *subjectivity* — different operators may assign different grades to the same fruit; (ii) *fatigue* — accuracy drops significantly during extended shifts; and (iii) *lack of traceability* — no digital record is created for each decision.

Instrumental methods such as colourimeters and refractometers offer objective measurements but are destructive and unsuitable for high-throughput conveyor lines. This has driven interest in *non-destructive, vision-based* inspection systems that can capture and classify hundreds of fruits per minute [7].

Justification. The limitations above motivate our choice of an *automated, camera-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

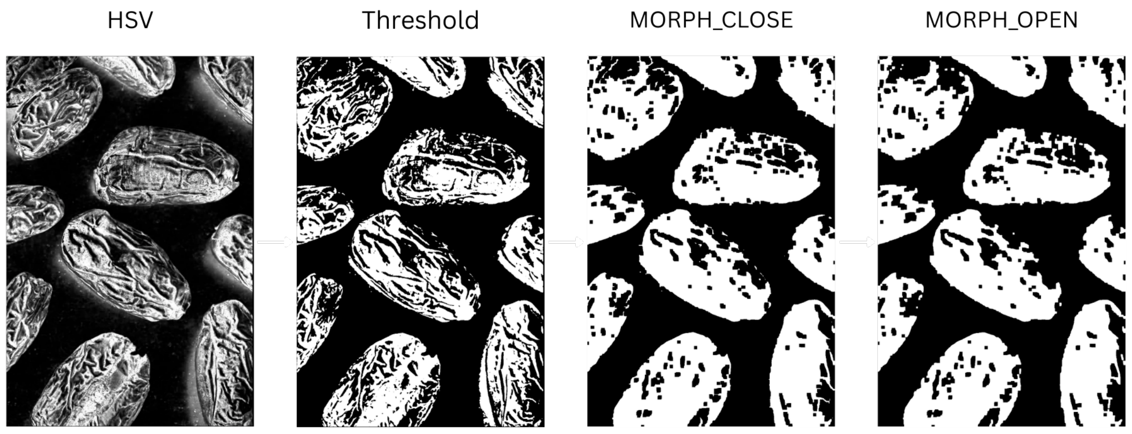
Thresholding converts a grey-scale image into a binary mask. Fixed (global) thresholds fail when lighting conditions vary. **Otsu's method** [8] computes the optimal threshold automatically by maximising the inter-class variance between foreground and background pixels, making it adaptive to each frame.

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 deployment strategy uses **Docker** containers [12] to encapsulate the Python runtime, TensorFlow CPU, and the trained model into a single reproducible image. This ensures that the API behaves identically in development and production, eliminating "it works on my machine" issues.

2.5 Generative AI for Operational Decision Support

Large Language Models (LLMs) such as GPT and Gemini [13] 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. [14]	Dates	YOLO	YOLO + K-Means	Yes	No	No
Almutairi et al. [15]	Dates	YOLOv8	YOLOv8	Yes	No	No
Lipiński et al. [16]	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 $< 2\,\mathrm{s}$ 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.

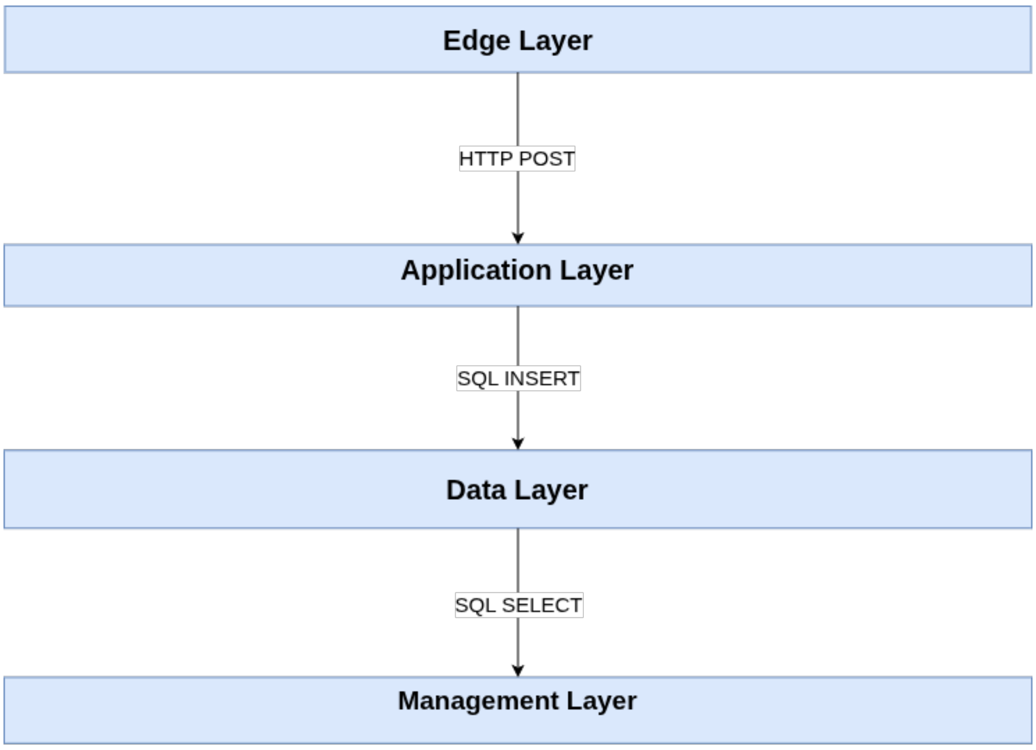


Figure 3.1: High-level four-layer architecture of the proposed system.

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 PLACEHOLDER]

API request lifecycle (POST → detect → classify → respond → log).

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 (%)

[FIGURE PLACEHOLDER]

Screenshot of the Supabase `logs` table with sample data.

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`).

```python
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`).

```python
# 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.

```dockerfile
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 [12].
- **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 PLACEHOLDER]
Render dashboard showing the deployed API service.

Figure 3.5: Render deployment dashboard for the cloud API.

Chapter 4

Intelligent Preprocessing and Object Detection

4.1 Overview of the Two-Stage Pipeline

4.2 Image Decoding and Colour-Space Transformation

4.3 Foreground Mask Generation

4.3.1 Gaussian Blur for Noise Reduction

4.3.2 Otsu's Adaptive Thresholding

4.3.3 Morphological Closing and Opening

4.4 Scene-Aware Adaptive Logic

4.4.1 Macro Mode (Single Object, Coverage $>$ 50%)

4.4.2 Conveyor Mode (Multiple Objects)

4.5 Artifact Rejection

4.6 Algorithm Summary

Chapter 5

Dataset Preparation and Model Training

5.1 Dataset Description

5.2 Data Reorganisation

5.3 Train / Test Split

5.4 Data Loading and Augmentation

5.5 Model Architecture: MobileNetV2

5.5.1 Inverted Residual Blocks and Depthwise Separable Convolutions

5.5.2 Transfer Learning Strategy

5.6 Training Configuration

5.7 Model Serialisation

Chapter 6

Experimentation and Results

6.1 Experimental Setup

6.2 Training Results

6.2.1 Accuracy and Loss Curves

6.2.2 Convergence Analysis

6.3 Test-Set Evaluation

6.3.1 Confusion Matrix

6.3.2 Precision, Recall, and F1-Score

6.3.3 Discussion of Misclassifications

6.4 Preprocessing Pipeline Evaluation

6.5 End-to-End System Performance

6.6 Generative AI Report Quality

6.7 Comparison with Related Work

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

[13] 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

[14] 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.

[15] 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.

[16] 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.

Appendix A

Source Code Listings

Appendix B

Dockerfile and Environment Configuration

Appendix C

Sample Generative AI Quality Report

Appendix D

Dataset Sample Images