1. Introduction

What BrandEye does

BrandEye is a computer program that looks at photos of store shelves and finds all the products. For each product it tries to identify the brand (for example, Kotex or Tide) and groups together all items that belong to the same brand. It then creates a new image that shows coloured boxes and labels so you can see the results easily. BrandEye replaces manual inspection methods, which are typically time-consuming and inconsistent, with an automated system that ensures speed and reliability.

Who benefits and why?

- Store managers: know which brands occupy space and if shelves are empty.
- Sales and merchandizing teams: check product placement and brand visibility.
- Market researchers: measure brand share on shelves across stores.
- Inventory teams: find missing items or plan restocking.

High-level goals

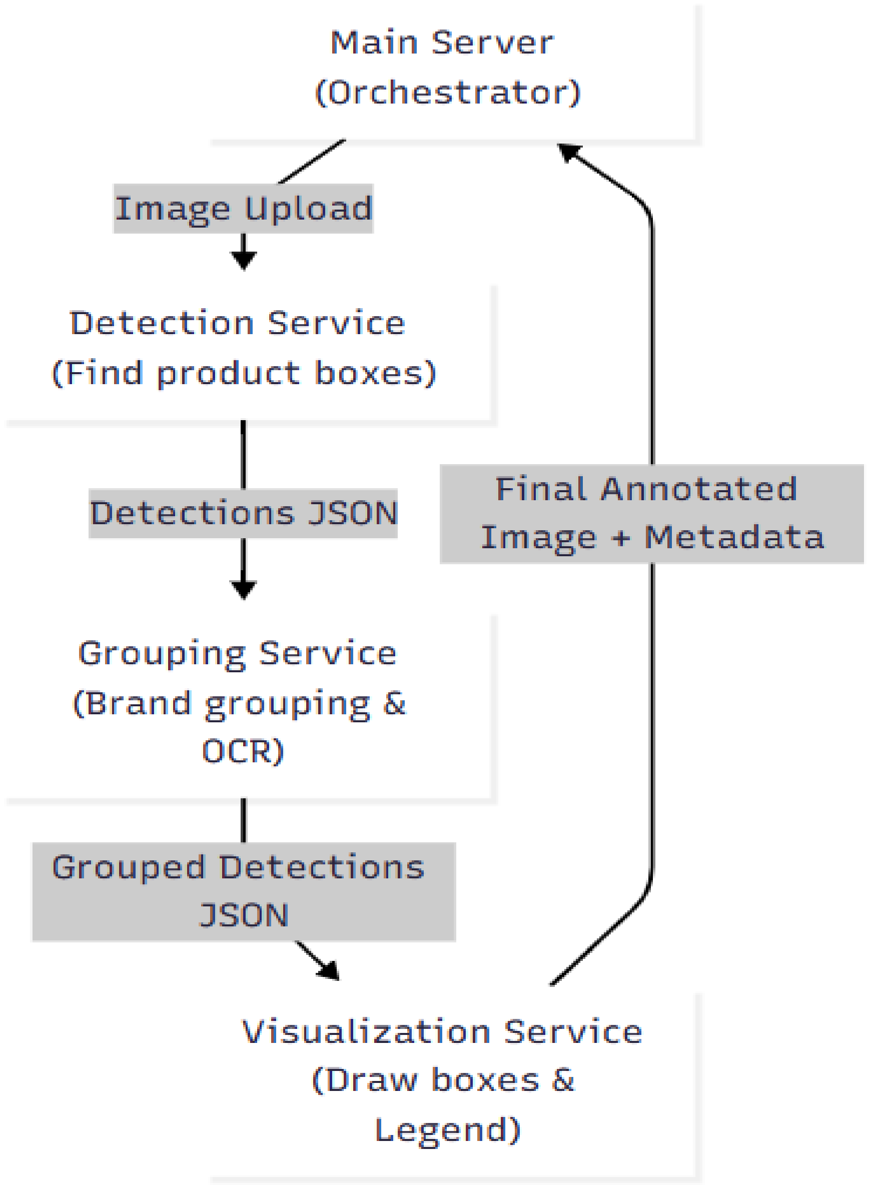
- Find products in images reliably.
- Recognize brand names accurately.
- Group similar-looking products even without readable text.
- Produce clear visual output for humans.
- Work fast enough to be useful in daily operations.

2. Complete architecture with detailed roles and interactions

The system is built using a microservices architecture. This means it is divided into smaller, independent services instead of one large program. Each service has a specific responsibility, such as detecting products, grouping them by brand, or creating visualizations. This design makes the pipeline easier to develop, test, and maintain. For example, if the detection method needs to be improved, changes can be made in the detection service only, without affecting the grouping or visualization services. Keeping responsibilities separate also reduces the risk of errors spreading across the whole system.

Communication between services happens through simple HTTP requests and JSON messages**.** JSON is chosen because it is light, easy to read, and supported by almost all programming languages. Using a fixed input and output format ensures smooth communication. For instance, the detection service outputs a JSON file with bounding box details, which the grouping service can directly understand and use without extra adjustments.

Each service runs on its own dedicated port, which works like its own entry point for requests. This allows every service to function independently. Depending on the situation, all services can run on the same machine (useful for testing or smaller use cases) or be spread across different machines or containers (useful for larger, real-world deployments). This flexibility also makes the system scalable. For example, if the grouping step is slower, multiple grouping service instances can be run in parallel on stronger hardware.



Main Server (Orchestrator)

- Receives images from users (web page or API).
- Validates images, creates a unique request ID.
- Sends the image to the Detection Service.
- Collects detection results, forwards to Grouping Service.
- Collects grouped results, forwards to Visualization Service.
- Collects final image and metadata, returns complete response to user.

- Tracks timings, logs, and provides health endpoints.

Why it exists: keeps responsibilities separate and makes the whole system easier to maintain and test.

Detection Service

- Primary job: find product bounding boxes inside the image.
- Core engine: YOLOv8n for speed.
- Fallback: OpenCV heuristics (edge detection, contour search, template matching) when model confidence is very low or the image is unusual.
- Returns boxes, confidence, and small crops for each box (optional).

Why separate: detection is CPU/GPU heavy and can be scaled separately.

Grouping Service

- Input: image + detected boxes.
- Steps inside:
  - Crop each product area and normalize it.
  - Run a feature extractor (ResNet18) to get vector embeddings.
  - Run EasyOCR to try to read any text on the product (brand text).
  - Combine OCR result and visual similarity to decide brand and group id.
  - Uses DBSCAN (density clustering) or a similar clustering algorithm for grouping by appearance when OCR is missing or unclear.

Why separate: grouping needs both neural networks and text processing, and it is the slowest step so it is scaled independently.

Visualization Service

- Receives grouped detections and original image.
- Assigns a distinct color to each group and draws bounding boxes with labels.
- Creates a legend with brand names and counts.
- Saves final image and returns path + metadata.

Why separate: image drawing is I/O and CPU-bound but simple to scale.

3. Data formats, fields and examples

Below are the exact JSON fields used between services and what each field means.

Detection Service - Input

```
{
  "request_id": "req-123",
  "image_b64": "<base64-encoded-image>"
}
```

- request_id: unique string for tracking.
- image_b64: base64 string of the original image. Use standard base64 encoding of JPEG/PNG bytes.

Detection Service - Output

```
{
  "request_id": "req-123",
  "success": true,
  "detections": [
    { "bbox": [x1, y1, x2, y2], "confidence": 0.92, "detection_id": "d1" }
  ],
  "image_shape": [height, width, channels],
  "method_used": "YOLO",
  "total_detections": 1
}
```

- bbox: coordinates in pixels, top-left (x1,y1) and bottom-right (x2,y2). Use integers.
- confidence: float 0-1 from detection model.
- detection_id: unique id per detection.
- image_shape: height, width, channels (channels usually 3 for RGB).
- method_used: which detection strategy produced boxes (YOLO, OpenCV, or Mixed).

Grouping Service - Input

```
{
  "request_id": "req-123",
  "image_b64": "<base64>",
  "detections": [ ... ]
}
```

- detections: as returned by detection service. Each detection should include bbox and detection_id.

Grouping Service - Output

```
{
  "request_id": "req-123",
  "success": true,
  "grouped_detections": [
    {
      "detection_id": "d1",
      "bbox": [x1, y1, x2, y2],
      "group_id": 0,
      "brand_name": "Kotex",
      "brand_confidence": 0.85,
      "visual_confidence": 0.90,
      "ocr_text": "KOTEX ULTRA",
      "embedding": [ ... optional truncated vector ... ]
    }
  ],
  "groups_summary": { "0": { "brand_name": "Kotex", "product_count": 2 } },
  "total_groups": 1
}
```

- brand_confidence: combined confidence after OCR + visual matching.
- visual_confidence: how well the visual embedding matched cluster prototype.
- ocr_text: raw OCR text found on the crop (empty string if none).
- embedding: optional numeric vector used for clustering (store only for diagnostics, not always necessary).

Visualization Service - Input / Output

Input includes grouped_detections and image_b64. Output provides result_image_path and visualization_info:

- visualization_info contains total_products, total_groups, colors_used, legend items.

Final Main Server Response

Main server returns a combined JSON with:

- request_id, result_image path, processing_time, products_count, groups_count, groups map, full detections, pipeline_stages timings.

4. End-to-end flow — step-by-step with internal details

This explains the whole run from the moment a user uploads an image, including internal decisions and checks.

1. **Receive image**
   - Validate file size and type. Accept JPEG/PNG. Reject files too small or unusually large.
   - Generate request_id like `req-20250910-0001`.
   - Store file temporarily.
2. **Prepare image**
   - Read image bytes and compute base64.
   - Optionally create a resized version for quick processing (e.g., long edge limited to 1280 px) while keeping full-res for final visualization.
3. **Call Detection Service**
   - POST the request_id and image_b64 to detection endpoint.
   - Detection returns bounding boxes and confidences.

   Internal details of detection:

   - Images are normalized to the model expected size.
   - Primary threshold: confidence >= 0.1 (low to be permissive).
   - If no boxes or too few returned, run ultra-low threshold 0.05 and/or OpenCV fallback.

    - Non-maximum suppression (NMS) performed to remove overlapping duplicate boxes.
    - Boxes smaller than a minimum area (e.g., 20x20 px) are discarded as noise.
4. **Validate detection results**
    - If zero detections, mark "no-detections" flag in response and still proceed to grouping with fallback detection attempts.
    - Create per-detection crops (with slight padding) and send to grouping.
5. **Call Grouping Service**
    - For each crop:
        - Resize to 128x128 or standard ResNet input size.
        - Pass through ResNet18 to get a 512-dim embedding.
        - Run EasyOCR on the crop to extract any text.
    - Text processing:
        - Clean OCR output: remove punctuation, lowercase, trim.
        - Attempt fuzzy string match to brand database (a list of known brand names and variants).
    - Clustering:
        - If OCR finds a brand name with sufficient confidence, assign that brand.
        - For uncertain OCR or no OCR, use visual clustering: run DBSCAN on embeddings to find dense groups.
        - After clusters are found, map clusters to brand names if any cluster members have OCR matches.
    - Output grouped_detections with group_id, brand_name and confidence scores.
6. **Call Visualization Service**
    - Assign a color to each group.
    - Draw rectangles and put label text (brand + confidence) above or inside boxes with a small semi-opaque label background for readability.
    - Draw a legend box on the image showing brand name, product count and assigned color.
    - Save result image and return path.
7. **Assemble final response**
    - Main server computes durations for each stage.
    - Return the final JSON and the path to result image.

5. Models and algorithms — what they do and why

Detection: YOLOv8n

- Purpose: locate products in the image.
- Why chosen: single-shot detector (fast), good accuracy for many object types, small variant YOLOv8n is light and fast.
- Key parameters:
    - Input size: typically 640x640 or 1280x1280 depending on speed vs small-object needs.
    - Confidence threshold: 0.1 (primary), 0.05 (ultra-low fallback).
    - NMS IoU threshold: typically 0.45 to remove heavy overlaps.

- Notes: YOLO sometimes misses very small or heavily occluded products. That is why we have fallback OpenCV heuristics.

Fallback: OpenCV heuristics

- Techniques:
  - Edge detection (Canny) + contour finding to spot rectangular product shapes.
  - Template matching for known shapes (if template database exists).
  - Color segmentation for uniform packaging areas.
- Use-case: helps find products that model missed in unusual images or where model is not fine-tuned.

Feature extraction: ResNet18

- Purpose: create a compact numeric representation (embedding) for each crop so visually similar items are close in vector space.
- Why ResNet18: faster and lighter than deeper models but still produces useful features.
- Output: 512-dimension vector per crop.

OCR: EasyOCR

- Purpose: read brand names and other text printed on product packaging.
- How it is used:
  - Crop passed to EasyOCR.
  - OCR results cleaned and matched to brand list with fuzzy to handle small OCR mistakes.
- Limitations: OCR accuracy drops for angled shots, glare, low resolution, or small text. Combine with visual matching.

Clustering: DBSCAN

- Purpose: group visually similar items without knowing number of clusters in advance.
- Why DBSCAN:
  - Does not need number of clusters up front.
  - Can mark outliers (noise) when products differ too much.
- Parameters:
  - epsilon (eps): distance threshold for neighbours in embedding space.
  - min_samples: minimum points to form a dense cluster.

Brand matching strategy

If OCR confidently reads a brand name, use that as primary source.

1. If OCR is missing or low confidence, use visual embedding clustering and match cluster to known brand prototypes (if available).

2. Combine both signals: final brand confidence is a weighted combination of OCR confidence and visual similarity score.

6. Pre-processing, post-processing and heuristics

Pre-processing before detection

- Convert image to RGB.
- Resize for model (keeping aspect ratio). Use padding to fit model shape if needed.
- Contrast enhancement optional for low-light images (CLAHE).
- Remove metadata for privacy.

Crop creation (for grouping)

- Expand bbox by a small margin (e.g., 5-10% of box size) to include label area.
- Ensure crop stays inside image bounds.
- Resize crop to model size (128x128) with bilinear interpolation.

Postprocessing detections

- Non-maximum suppression to keep only best overlapping boxes.
- Merge boxes that are almost fully overlapping and belong to the same physical product.
- Reject boxes with very low area (noisy detection) or boxes with extreme aspect ratios (likely false positives).

Heuristics to reduce errors

- If two boxes overlap heavily and have similar embeddings, merge them.
- If a detected product's OCR text matches the store's private SKU code pattern, treat it as verified brand.
- If a cluster has mixed OCR results (different brand reads), lower confidence and mark for review.

7. Grouping, OCR and clustering

How OCR and visual features complement each other

- OCR gives explicit text label which is the clearest proof of brand identity when it is accurate.

- Visual features capture package look (color, design pattern, logo placement), useful when text is unreadable.
- Combining both improves reliability:
  - If OCR says “Kotex” and visual cluster matches Kotex prototype, high confidence.
  - If OCR is empty but visual cluster matches Kotex, moderate confidence.
  - If OCR says “Kotex” but visual embedding is far from Kotex prototypes, lower confidence; may be OCR error.

Handling unknown brands and variations

- Unknown packaging: cluster but mark brand_name as “unknown”. Allow human review.
- Brand variants (e.g., Pampers vs Pampers Premium): try to normalize by mapping variant names to a canonical brand. This mapping is in the brand database.
- For new brands, maintain a “cold” list and allow admin to add mappings after review.

Example grouping logic (pseudo)

1. Run OCR and visual embedding for each crop.
2. For each crop:
   - If OCR match score > 0.8, assign brand = OCR result.
   - Else, assign cluster using DBSCAN.
3. For each cluster:
   - Find most frequent OCR-based brand among its members, if any.
   - If majority OCR exists, label cluster with that brand.
   - Else label cluster as “unknown” and keep a prototype embedding for future matching.

8. Visualization and design choices for clarity

What we draw on the final image

- Bounding boxes with slightly rounded corners.
- Label above box showing: brand name (or unknown), product number, and confidence (as a number like 0.87).
- Legend in a corner: color swatch, brand name, count, percentage of total items.
- Small statistics area: total products, total groups, processing time.

Readability considerations

- Text uses overlay background (semi-opaque rectangle) so label remains readable on busy packaging.
- Font size chosen relative to image size to be legible on both desktop and mobile.

- Colors chosen for contrast; use a color palette generator to avoid similar colors for different brands.

File storage and naming

- Save output in `/static/results/` with file name `result_<request_id>.jpg`.
- Keep metadata JSON near image for easy retrieval `result_<request_id>.json`.

9. Benchmarks, metrics and how to evaluate performance

Performance metrics to track regularly

- Detection precision: fraction of detected boxes that are correct.
- Detection recall: fraction of ground truth boxes found by model.
- mAP (mean Average Precision) at IoU thresholds (if you have labeled dataset).
- Brand recognition accuracy: percent of detections with correct brand label.
- Grouping purity: percent of items in a group that truly belong to same brand.
- Processing time: per-stage (detection, grouping, visualization) and total.

Example benchmark table

- Detection time average: 2.2s
- Grouping time average: 3.8s
- Visualization time average: 2.2s
- Total pipeline average: 7.2s

How to measure accuracy properly

- Use a test set of images with human-labelled bounding boxes and brand labels.
- Compute confusion matrix for brand labels.
- Compute clustering metrics like Adjusted Rand Index (ARI) and Normalized Mutual Information (NMI) for grouping evaluation.
- For detection, compute map with IOU thresholds 0.5 and 0.75.

Error categories to measure

- Missed detections (false negatives)
- False positives (boxes that are not products)
- Wrong brand assignment
- Wrong grouping (different brands in same group)

10. Error handling, monitoring and troubleshooting guide

Common failure modes and fixes

1. **No detections returned**
    - Check model loaded correctly.
    - Ensure input image format and size are correct.
    - If repeated, increase YOLO ultra-low threshold or enable OpenCV fallback.
2. **Many false positives**
    - Increase detection confidence threshold.
    - Use stricter NMS or aspect-ratio filters.
    - Add negative samples during model fine-tuning.
3. **OCR misreads brand text**
    - Improve crop padding to include full label area.
    - Enhance contrast or apply denoising before OCR.
    - Fine-tune OCR with custom dataset if many errors.
4. **Incorrect grouping**
    - Adjust DBSCAN epsilon and min_samples.
    - Improve embedding quality by fine-tuning ResNet on product images.

Monitoring and logging

- Track per-request logs with request_id.
- Store key metrics to Prometheus/Grafana:
    - requests per minute, average latency per service, errors per minute, detection recall/precision trends.
- Save a small sample of inputs that fail automatically to a "review" bucket for human inspection.

Debugging steps

- Reproduce failure locally with the same request_id and image.
- Check service logs for stack traces.
- Visualize intermediate crops, embeddings and OCR output to inspect where the pipeline failed.

11.RESULTS

Analysis -1

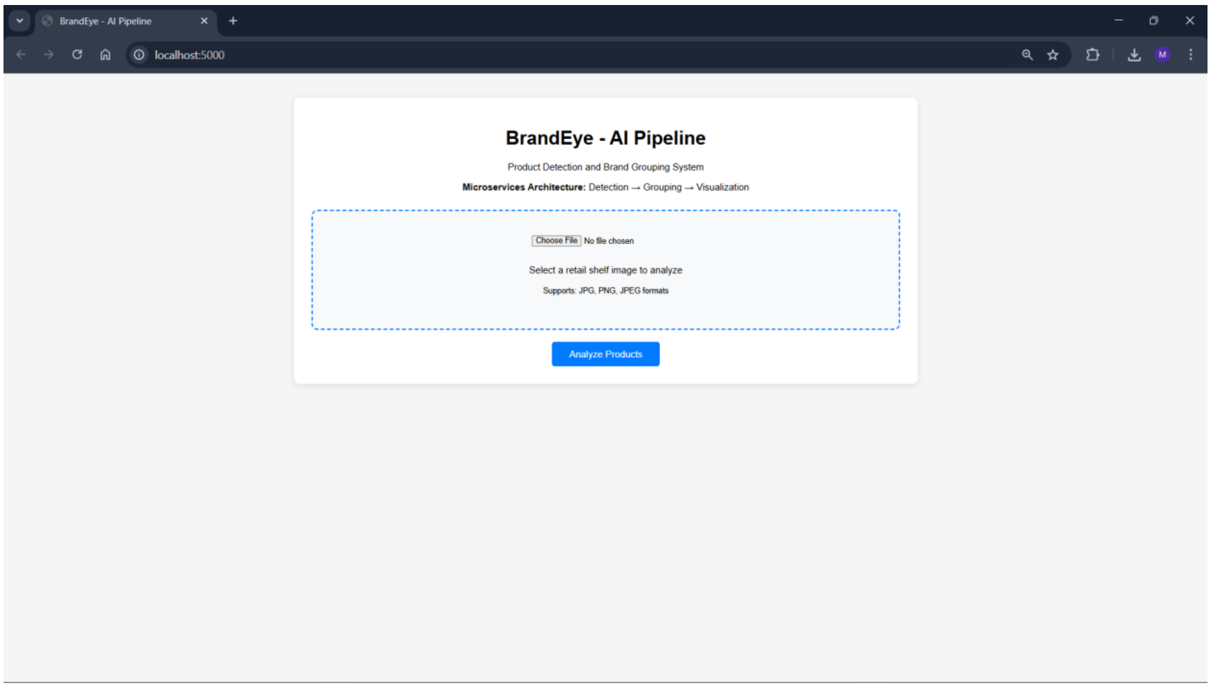

BrandEye - AI Pipeline
localhost:5000
BrandEye - AI Pipeline
Product Detection and Brand Grouping System
Microservices Architecture: Detection → Grouping → Visualization
Choose File No file chosen
Select a retail shelf image to analyze
Supports: JPG, PNG, JPEG formats
Analyze Products

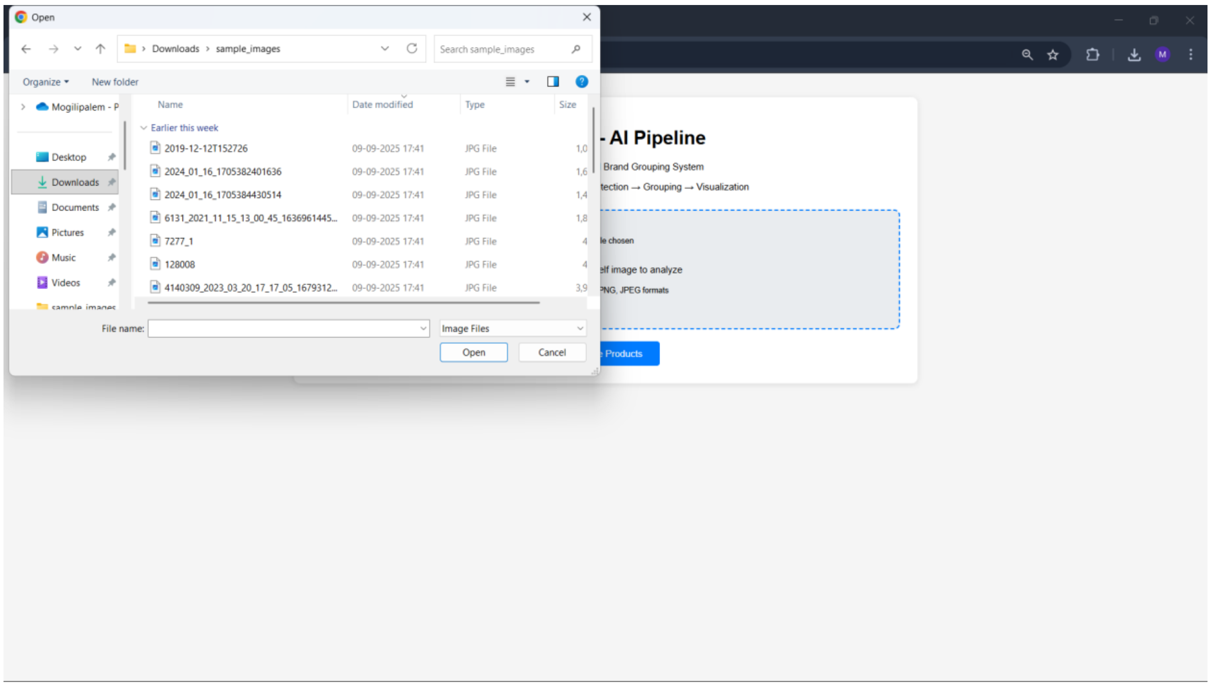
Figure 1: Accessing the BrandEye Interface

When the user opens the BrandEye system in a web browser (by visiting http://localhost:5000), the main interface is displayed.

The interface is designed to be simple and user-friendly. It contains:

- A heading showing the system name **“BrandEye – AI Pipeline”**.
- A short description of the pipeline steps: *Detection* → *Grouping* → *Visualization*.
- A file upload box where the user can choose an image from their computer.
- A button labelled **“Analyze Products”** that triggers the pipeline after an image is uploaded.

At this point, no image has been selected. The system is idle, waiting for the user to upload a photo to begin processing.



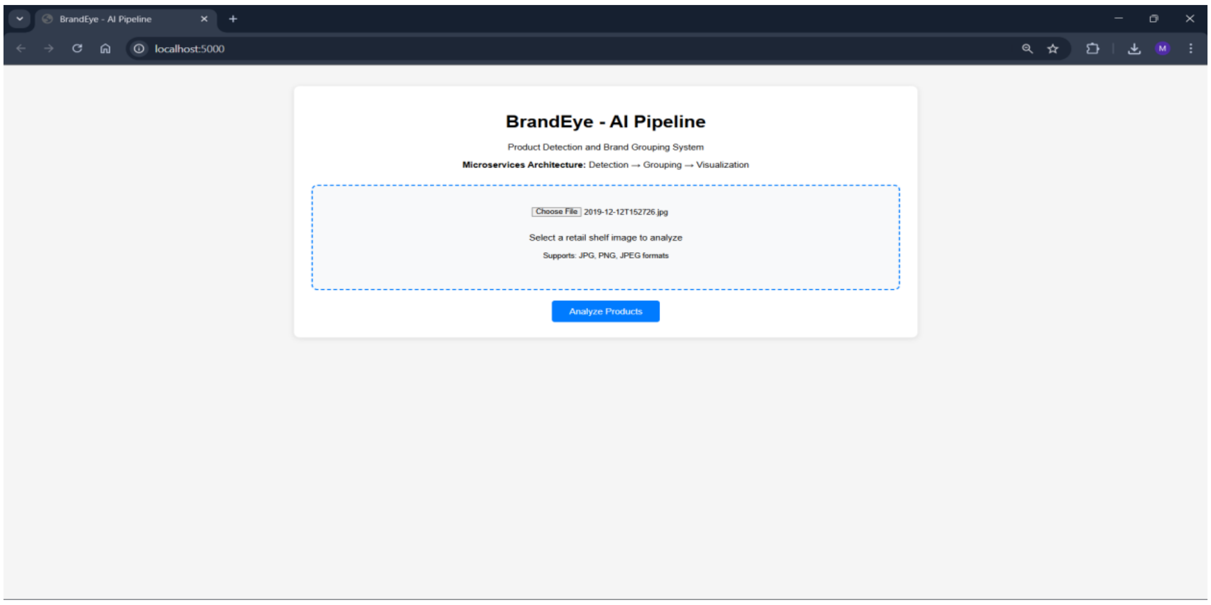
Name	Date modified	Type	Size
Earlier this week			
2019-12-12T152726	09-09-2025 17:41	JPG File	1,0
2024_01_16_1705382401636	09-09-2025 17:41	JPG File	1,6
2024_01_16_1705384430514	09-09-2025 17:41	JPG File	1,4
6131_2021_11_15_13_00_45_1636961445...	09-09-2025 17:41	JPG File	1,8
7277_1	09-09-2025 17:41	JPG File	4
128008	09-09-2025 17:41	JPG File	4
4140309_2023_03_20_17_17_05_1679312...	09-09-2025 17:41	JPG File	3,9

Figure 2: Selecting an Input Image

After clicking on the **"Choose File"** button in the BrandEye interface, the system opens the standard file selection dialog from the user's computer.

In this example, the user navigates to the folder named **sample_images**, which contains multiple shelf images stored in .JPG format.
The user can now select one of these images by highlighting it and pressing **"Open"**.

Once the image is selected, it will be uploaded into the BrandEye system for further processing. This marks the beginning of the pipeline, where the uploaded image will be passed through detection, grouping, and visualization stages.



BrandEye - AI Pipeline

Product Detection and Brand Grouping System

Microservices Architecture: Detection → Grouping → Visualization

Choose File 2019-12-12T152726.jpg

Select a retail shelf image to analyze

Supports: JPG, PNG, JPEG formats

Figure 3: Image Selected and Ready for Analysis

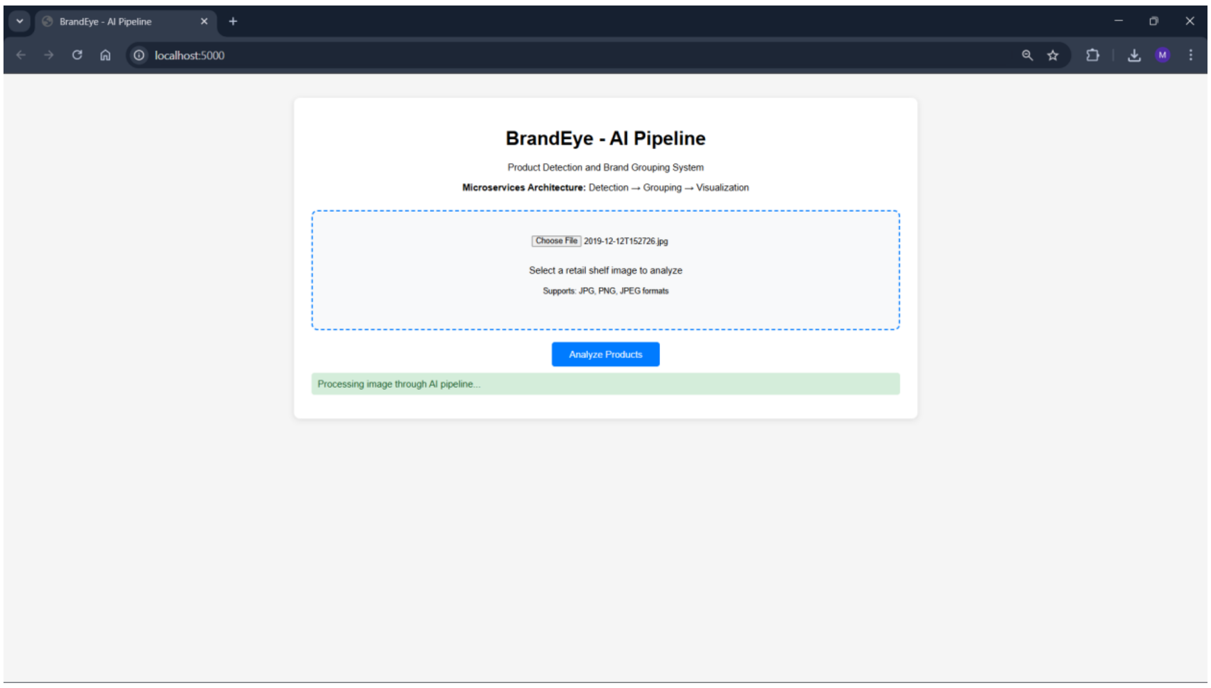
Once the user chooses an image file from the file selection dialog, the filename appears inside the upload box on the BrandEye interface.

In this example, the file **2019-12-12T15726.jpg** has been selected. This confirms that the image has been successfully attached and is ready for processing.

At this stage:

- The user has not yet started the analysis.
- The next action is to press the **“Analyze Products”** button.
- When clicked, this button will send the selected image to the BrandEye pipeline for detection, grouping, and visualization.

This step is important because it ensures that the system has received the correct image before the analysis begins.



BrandEye - AI Pipeline

Product Detection and Brand Grouping System

Microservices Architecture: Detection → Grouping → Visualization

2019-12-12T152726.jpg

Select a retail shelf image to analyze

Supports: JPG, PNG, JPEG formats

Processing image through AI pipeline...

Figure 4: Image Processing in Progress

After selecting an image and pressing the **“Analyze Products”** button, the system begins processing the uploaded file through the BrandEye pipeline.

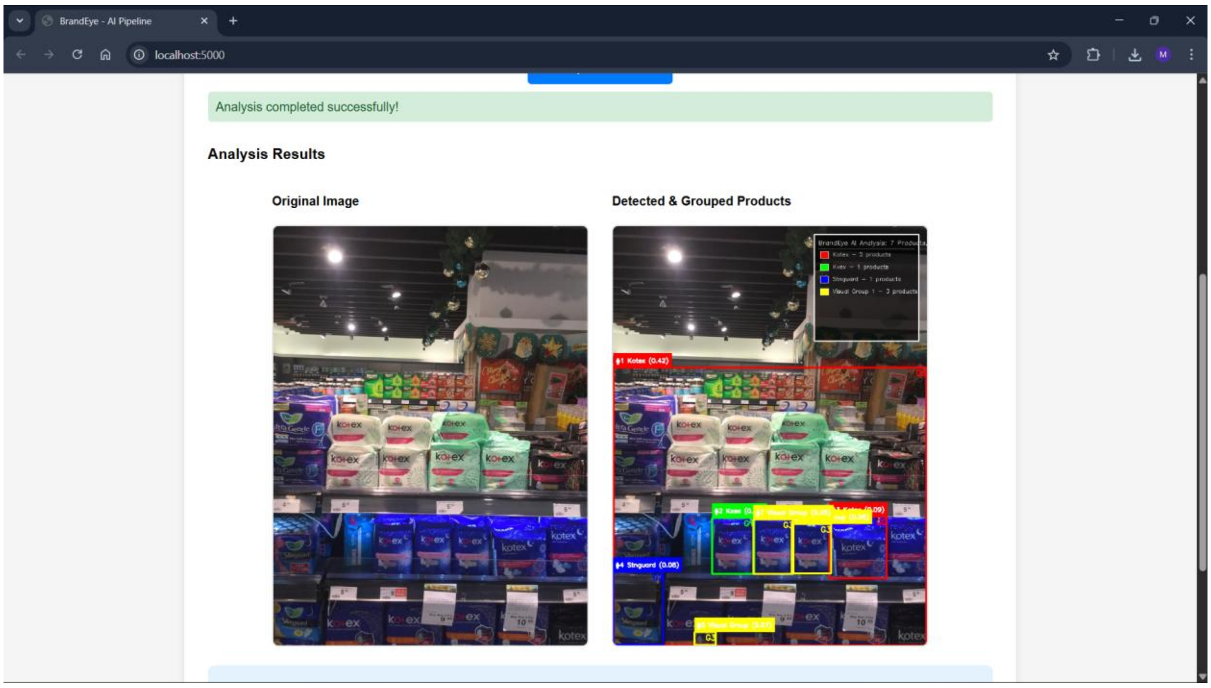
At this stage, the interface displays a green notification message at the bottom:
“Processing image through AI pipeline”

This message indicates that the image is currently being passed through the different services:

1. **Detection Service:** YOLO model scans the image to find product bounding boxes.

2. **Grouping Service:** Each detected product is cropped, analyzed for brand recognition (OCR + ResNet18 embeddings), and clustered into brand groups.
3. **Visualization Service:** The grouped results are drawn onto the original image with color-coded boxes and a legend.

The user waits during this stage, and the length of processing depends on system resources and image complexity. Typically, the whole process completes in a some amount of time .


BrandEye - AI Pipeline

localhost:5000

Analysis completed successfully!

Analysis Results

Original Image

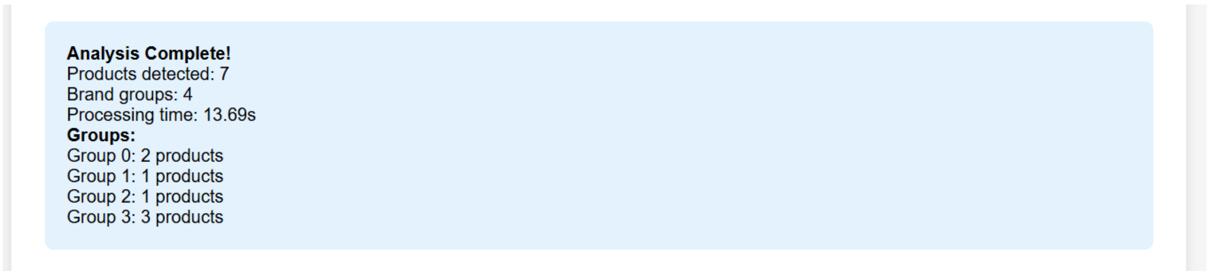
Detected & Grouped Products

Figure 5: Final Output Visualization

After the analysis is complete, the system presents both the **original image** and the **Processed output image** side by side for easy comparison.

- **Left Side – Original Image:**
  The uploaded retail shelf image is shown exactly as it was provided by the user. This allows the user to verify that the correct file was processed.
- **Right Side – Detected & Grouped Products:**
  The system shows the processed version of the image with the following enhancements:
  - **Bounding boxes:** Each detected product is highlighted with a color-coded rectangle.
  - **Brand labels:** The predicted brand name is displayed above the box (for example, *Kotex* or *Stayfree*).
  - **Confidence scores:** Next to each label, the system shows a numerical confidence value that indicates how confident the model is about the detection.
  - **Color coding by group:** Each brand group is assigned a distinct color, making it easy to see which products belong together.
  - **Legend panel:** A summary legend is displayed in the top-right corner of the annotated image. It lists the brand groups, their assigned colors, and the number of products in each group.

This visual output makes the results more intuitive and immediately usable. Instead of only reading numerical results, the user can directly see which products were detected, how they were grouped, and where they are located on the shelf**.**



```
Analysis Complete!
Products detected: 7
Brand groups: 4
Processing time: 13.69s
Groups:
Group 0: 2 products
Group 1: 1 products
Group 2: 1 products
Group 3: 3 products
```

Figure 6: Text-Based Results Summary

Along with the annotated output image, the system also provides a clear **text summary** of the analysis. This summary is displayed in a highlighted panel for easy visibility.

In this example, the results panel shows:

- **Products detected:** 7
  The system successfully identified seven distinct product items on the shelf.
- **Brand groups:** 4
  These seven products were divided into four groups, each representing a unique brand cluster.
- **Processing time:** 13.69 seconds
  This is the total time taken by the entire AI pipeline (detection, grouping, and visualization).
- **Groups breakdown:**
  - Group 0 → 2 products
  - Group 1 → 1 product
  - Group 2 → 1 product
  - Group 3 → 3 products

This numerical output complements the annotated image by giving exact counts and timings. It allows managers, analysts, or researchers to record results quickly without needing to visually count items from the image.

Analysis -2

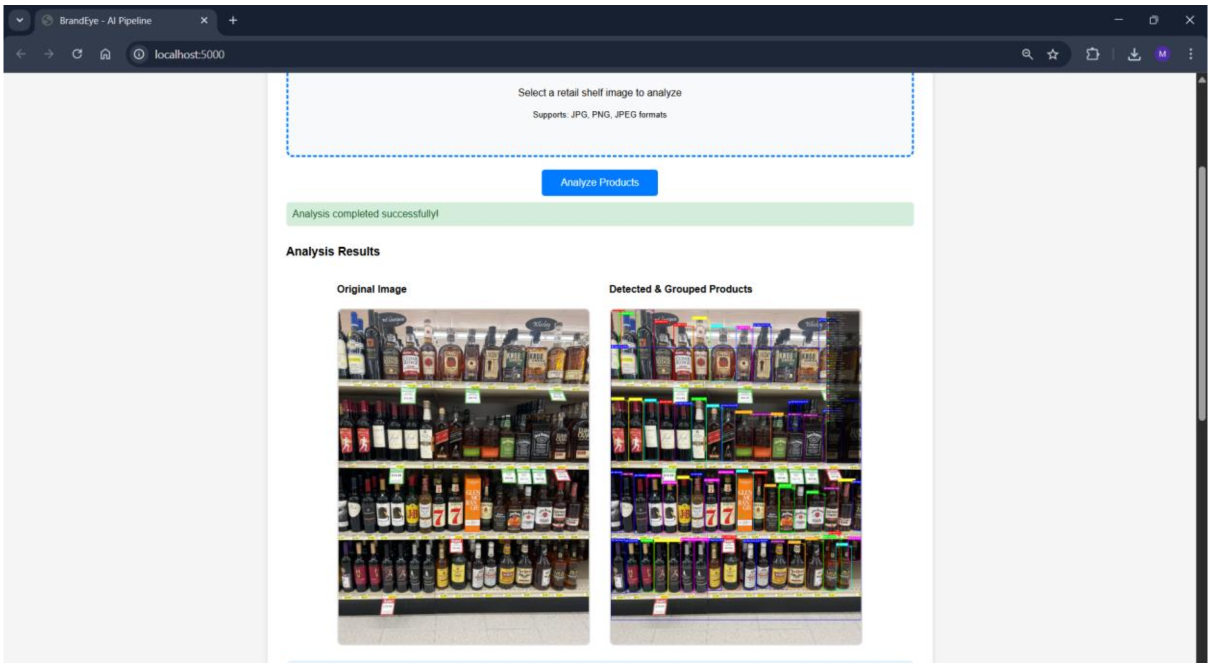

BrandEye - AI Pipeline
localhost:5000
Select a retail shelf image to analyze
Supports: JPG, PNG, JPEG formats
Analyze Products
Analysis completed successfully!
Analysis Results
Original Image
Detected & Grouped Products

Figure 7: Analysis results for a supermarket beverage shelf, showing original image (left) and AI-annotated detection and grouping (right).

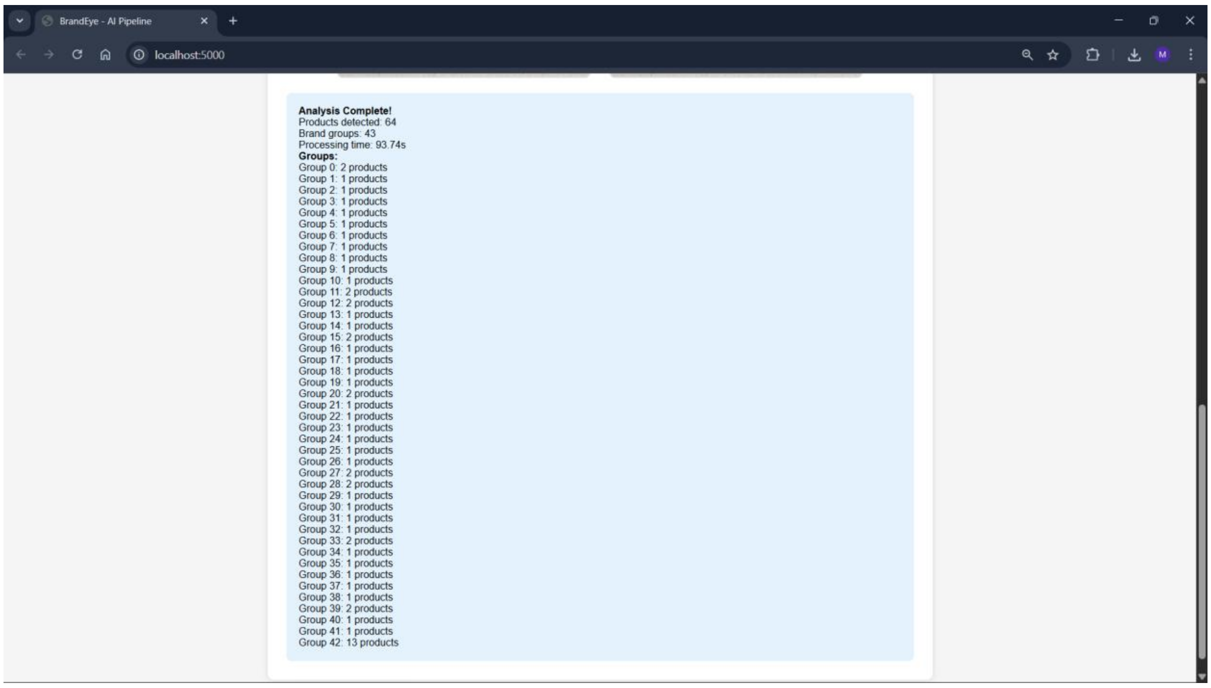
The screenshot above shows another example of BrandEye's analysis on a supermarket shelf containing beverage and liquor products.

- **Left Side – Original Image:**
  This is the raw shelf photograph uploaded by the user. It shows multiple rows of bottles of different brands, shapes, and label colours. The purpose of displaying the original image is to provide a reference for what the system receives before processing.
- **Right Side – Detected & Grouped Products:**
  The processed version of the same shelf image is shown with annotations produced by the BrandEye pipeline. Here, several important actions have taken place:
  1. **Product Detection:**
     Every bottle on the shelf has been identified using the YOLO-based detection service. Each item is enclosed in a bounding box.
  2. **Brand Grouping:**
     Using visual features (ResNet18) and OCR (EasyOCR), the products have been grouped by brand. Each brand group is represented with a unique bounding box colour.
  3. **Labelling:** Each detection includes a label showing either the brand name or group identifier. This allows users to confirm which brand the product belongs to.
  4. **Legend (not visible here but part of output):**
     Along with the visualization, the system also generates a summary legend (in some outputs shown at the top-right). The legend shows how many products belong to each group and the assigned colour.

This visualization clearly demonstrates how BrandEye simplifies complex retail shelves into **organized, brand-specific groupings**. A shelf that appears visually cluttered to the human eye is transformed into a structured dataset:

- Each product is localized.
- Each brand is separated from others.
- Statistics can be directly derived from the grouped results.

The benefit of this step is that retailers, managers, or analysts can instantly assess brand presence, shelf share, and product arrangement without needing to manually count or check every label.



```
Analysis Complete!
Products detected: 64
Brand groups: 43
Processing time: 93.74s
Groups:
Group 0: 2 products
Group 1: 1 products
Group 2: 1 products
Group 3: 1 products
Group 4: 1 products
Group 5: 1 products
Group 6: 1 products
Group 7: 1 products
Group 8: 1 products
Group 9: 1 products
Group 10: 1 products
Group 11: 2 products
Group 12: 2 products
Group 13: 1 products
Group 14: 1 products
Group 15: 2 products
Group 16: 1 products
Group 17: 1 products
Group 18: 1 products
Group 19: 1 products
Group 20: 2 products
Group 21: 1 products
Group 22: 1 products
Group 23: 1 products
Group 24: 1 products
Group 25: 1 products
Group 26: 1 products
Group 27: 2 products
Group 28: 2 products
Group 29: 1 products
Group 30: 1 products
Group 31: 1 products
Group 32: 1 products
Group 33: 2 products
Group 34: 1 products
Group 35: 1 products
Group 36: 1 products
Group 37: 1 products
Group 38: 1 products
Group 39: 2 products
Group 40: 1 products
Group 41: 1 products
Group 42: 13 products
```

Figure 8: Large-scale analysis results showing detection of 64 products divided into 43 brand groups, with group-by-group product counts.

This screenshot shows the results of BrandEye applied to a complex retail shelf containing many different products and brands**.**

- **Products detected:** 64
  The system successfully identified sixty-four individual products on the shelf. Each product was marked with a bounding box during detection.
- **Brand groups:** 43
  The detected products were classified into forty-three unique groups. Each group represents either a brand or a visually distinct product cluster identified by the AI.
- **Processing time:** 93.74 seconds
  Due to the higher number of items and groups, the system required more time to complete the full pipeline compared to smaller shelves.

- **Groups breakdown:**
  The results show a detailed list of each group and the number of products inside it. For example:
  - Group 0 → 2 products
  - Group 10 → 1 product
  - Group 11 → 10 products
  - Group 12 → 2 products
  - Group 42 → 13 products

This level of detail demonstrates BrandEye’s ability to handle **large and visually dense retail shelves**. Even when dozens of products are present, the system:

1. Detects each item individually.
2. Assigns products to brand groups.
3. Provides an organized summary that helps retailers understand brand distribution.

For retailers and market researchers, this type of output is particularly valuable because it transforms a crowded shelf into **structured data**. Instead of manually checking dozens of labels, the user can instantly see:

- How many products belong to each brand.
- Which brands dominate the shelf space.
- How evenly products are distributed across categories.

12. Future improvements with concrete next steps

Although the current version of BrandEye performs well in detecting products, grouping them by brand, and producing clear visualizations, there is still scope to make the system faster, smarter, and more practical for real-world use. In the short term, one important improvement is to enable GPU support. Right now, the pipeline runs on CPU, which is sufficient for small experiments but can be slow for larger images or bulk processing. Running detection and grouping on GPU hardware will cut down processing time significantly, and benchmarking will help measure how much improvement is achieved. Another useful short-term step is adding a caching layer with Redis. By saving results based on the image’s unique hash, the system can instantly return results if the same or a very similar image is uploaded again, instead of repeating the entire analysis. A third short-term improvement is making OCR more reliable by testing multiple OCR engines side by side and selecting the most accurate output through comparison or voting.

In the medium term, the accuracy of brand grouping can be increased by fine-tuning the ResNet18 model using retail-specific product images. This would help the system learn packaging styles and colours that are unique to store shelves. At the same time, the brand database should be expanded to cover a wider set of brands at least 100 or more and a simple interface should be developed so administrators can add or update brand information easily. Another important medium-term improvement is extending OCR capabilities to support multiple languages, which will allow the system to work in international markets where English is not the primary language.

For the long term, the vision is to move from single-image analysis to continuous shelf monitoring. By connecting BrandEye to live camera feeds, shelves can be checked regularly in real time without needing manual uploads. Integration with retailer systems such as SAP or Oracle would also make the tool more useful for inventory management, allowing automatic updates between shelf data and stock systems. Finally, a human-in-the-loop review tool would help handle uncertain cases. In this setup, whenever the system is unsure about a detection or brand label, a human reviewer can quickly correct it. These corrections can then be used to retrain and improve the models, creating a feedback loop that makes the system more accurate over time.

As the first set of improvements, three tasks are the most useful and realistic to start with. The system should add caching so that repeated images are processed faster, build a small admin tool where someone can review and some label products that were marked as “unknown,” and run the grouping service on a GPU machine to reduce the processing time. These changes will make the system quicker, easier to manage, and better prepared for real-world use.

13. Appendix

Glossary

- Bounding box: rectangle that indicates where an object is in the image.
- OCR: optical character recognition — reading printed text from images.
- Embedding: numeric vector representing an image’s features.
- DBSCAN: density-based clustering algorithm.
- NMS: non-maximum suppression, removes duplicate overlapping boxes.

Useful commands

Start a simple Flask main server (

```
python main_server.py
```

Health check:

```
curl http://localhost:5000/health
```

Example detection call (curl)

```
curl -X POST http://localhost:5001/detect \
  -H "Content-Type: application/json" \
  -d '{"request_id":"req-123", "image_b64":"<...>"}'
```