

TEAM MEMBERS:-

Anant S Kumar	SE22UCAM002
Lalitya Yelavarthi	SE22UARI188
Venkatesh Vishnudas	SE22UECM053
Abhiram Ayla	SE22UARI005

1. Problem statement

The assignment is to build an **automatic bin-level SKU verification system** for a warehouse / e-commerce setting.

For each storage bin we have:

- A **bin image** (`bin-images/XXXX.jpg`)
- A **metadata JSON** (`metadata/XXXX.json`) listing:
 - ASIN / SKU
 - Product title
 - Expected quantity in that bin

The system:

1. Detects all products visible in the bin image.
2. Match detections to the SKUs from metadata.
3. Estimate per-SKU quantity from detections.
4. Compare predicted vs expected quantities and classify each SKU as:
 - **FULL_MATCH** – correct or nearly correct count
 - **PARTIAL_OCCLUDED** – under-count, likely due to occlusion
 - **NOT_FOUND** – SKU expected but not detected
 - **OVERCOUNT** – more items detected than expected (potential false positives)

On top of this, the project includes:

- Full **EDA** on images, metadata and model outputs.
 - A sequence of **pipeline versions** (v1–v5) with a comparison study.
 - A small **Streamlit app** that behaves like a “smart invoice / order validator” on top of the model outputs.
-

2. Data and repository structure

```
applied-ai-assignment1/
├── app/
│   ├── __init__.py
│   └── streamlit_app.py          # Streamlit UI (main app
entrypoint)

├── bin-images/                  # Bin photos (00001.jpg,
00002.jpg, ...)
│   └── ... (data, usually not all in GitHub)

└── legacy_pipelines/
    ├── __init__.py
    ├── validate_full_pipeline_v2.py  # Older baseline pipeline
    └── validate_full_pipeline_v4_clip018.py

├── metadata/                   # Per-bin JSON metadata
(00001.json, ...)

└── mlops/
    ├── __init__.py
    ├── evaluate_model_v3_counts.py  # Evaluation + count-style
metrics
    └── plot_results.py            # CLI plotting for metrics

└── models/
```

```

|   ┌─ __init__.py
|   └─ validate_full_pipeline_v5.py    # Final CV pipeline (v5:
BOX=0.17, CLIP=0.20)

└─ notebooks/                         # EDA / experimentation notebooks

  ┌─ out/
  |   ├─ summary_*.csv                # Evaluation summaries (v1–v5)
  |   ├─ *_detected_*.jpg            # Visualisations with drawn boxes
  |   └─ inference_log.csv (optional)

  ┌─ weights/
  |   ├─ GroundingDINO_SwinT_OGC.py  # GroundingDINO config
  |   └─ groundingdino_swint_ogc.pth  # GroundingDINO weights (large,
may be local only)

  ┌─ .gitignore
  ┌─ Assignment #1 [Applied AI for Industry] (1).pdf
  ┌─ README.md
  ┌─ evaluate_model.py               # Older eval script
  ┌─ evaluate_model_v2.py
  ┌─ plot_results.py                # Older plotting script
  ┌─ requirements.txt
  ┌─ s3_downloader.py               # Helper to fetch data from S3
  ┌─ sanity_check.py                # Quick sanity run script
  ┌─ validate_full_pipeline.py       # Very first strict prototype
  └─ validate_full_pipeline_v3_dino_loose.py # Intermediate “loose
DINO” version

```

- Each **JSON** has a **BIN_FCSKU_DATA** dict mapping ASIN → {name, quantity}.

The **summary CSVs** are one row per (bin, SKU) with:

`image_id, asin, sku_name, expected, visible, status`

•

- `expected` – quantity from metadata
- `visible` – quantity inferred from detections
- `status` – categorical evaluation label

Dataset size (for the main experiments):

- **~987–988 bins**
 - **2702–2704 SKU-bin pairs** depending on version (v2–v5).
-

3. Exploratory Data Analysis (EDA)

Implemented an `EDA.py` script that:

- Flattens all metadata JSONs into a single table
- Analyses image characteristics
- Analyses all `summary_*.csv` files
- Writes plots and CSVs into `out/eda/`

3.1 Metadata EDA

From the flattened metadata:

- Each row: `image_id`, `asin`, `sku_name`, `expected`
- Key findings (qualitative, from the script's outputs):
 - **SKUs per bin**: a moderate number (typical small bin with a handful of SKUs).
 - **Expected quantities**: vary from 1 to higher counts; most rows have small integers (1–6).
 - Some SKUs appear in **multiple bins** (popular products) – captured by `value_counts` on `asin`.

Plots produced:

- Histogram of **SKUs per bin** – helps understand workload per image.
- Histogram of **expected quantity per row** – shows whether the model must count many instances or just presence.

These support design choices:

- Because quantities are small, a **hard cap** on detections per SKU (expected + tolerance) is reasonable.
- Many bins have multiple SKUs, so a **multi-prompt** text-conditioned detector (GroundingDINO) fits well.

3.2 Image EDA

The script scans `bin-images` with Pillow and logs:

- Width, height and aspect ratio (w / h)
- Orientation (landscape / portrait / square)

Typical findings:

- Most images are similar resolution; aspect ratios lie around a narrow band.
- A mix of landscape and portrait bins, which justifies using **normalized bounding boxes** from GroundingDINO.

Plots:

- Histograms of **width, height, aspect ratio**
- Bar chart of **orientation counts**

These ensure there's no systematic anomaly (e.g. a subset of very tiny or corrupted images).

3.3 Summary / Model Output EDA

For each `summary_v*.csv` the EDA script prints:

- Total SKU rows and #bins
- Distribution of `status`
- Presence metrics treating “visible > 0” as “SKU detected”
- Scatter plots of `expected` vs `visible` and status bar charts
- An aggregated table `summary_presence_metrics.csv` comparing versions.

This EDA is the backbone for the comparison study in Section 6.

4. Methodology

4.1 Model choices

The system combines:

1. **GroundingDINO** (Swin-T OGC variant)
 - Text-conditioned object detector.
 - Takes natural language prompts of product names.

- Produces bounding boxes with text scores per region.
2. **OpenAI CLIP (ViT-B/32 variant)**
- Image–text embedding model.
 - Used as a second-stage verifier:
 - Crop each candidate box.
 - Compute similarity between crop and SKU text.
 - Reject boxes with low CLIP score.

GroundingDINO – open-vocabulary detection

- **Why:**
 - Traditional detectors (Faster R-CNN, YOLO, etc.) require **class-specific training**.
 - Our SKU set is large, long-tail, and defined only via text (product names) with no bounding box labels.
 - GroundingDINO is an **open-vocabulary detector**: you can detect any object described by text, without training a custom detector per SKU.
- **Role:**
 - Generates candidate regions in the bin that correspond to each product description.

CLIP – vision–language similarity

- **Why:**
 - GroundingDINO scores are sometimes noisy, especially for similar-looking packaging.
 - CLIP is trained on large-scale image–text pairs, giving strong semantic alignment between images and natural language.
- **Role:**
 - Acts as a second-stage **verifier**:
 - Rejects DINO boxes that do not semantically match the SKU text.
 - Provides additional robustness when product names are long or noisy.
- **Model used:**
 - **ViT-B/32** variant, running on CPU (or GPU if available).

This two-stage design (DINO → CLIP) is the key **methodological choice**: it leverages powerful pretrained models instead of training a task-specific network from scratch, which would require extensive labelled data.

4.2 Per-bin pipeline

All `validate_full_pipeline_v*.py` versions follow the same high-level structure:

1. **Load metadata** for a bin (`image_id.json`).
 - o Extract list of SKUs: (`asin, name, expected_qty`).
2. **Prepare text prompts** for GroundingDINO.
 - o Typically the product title, possibly shortened / aliased.
3. **Run GroundingDINO** with:
 - o `BOX_THRESHOLD` – minimum detector box score.
 - o `TEXT_THRESHOLD` – minimum text relevance.
 - o `IOU_THRESHOLD_PER_SKU` – NMS IOU for per-SKU de-duplication (0.55 in all versions).
4. **Run CLIP verification**:
 - o For each candidate box and SKU text, compute cosine similarity.
 - o Keep those with similarity $\geq \text{CLIP_THRESHOLD}$.
5. **Count per SKU**:
 - o Group remaining boxes by ASIN.
 - o Apply any version-specific caps / tolerance.
 - o Compute `visible` count for each SKU.

Classify status for each SKU row:

```
if visible == 0:  
    status = "NOT_FOUND"  
elif abs(visible - expected) <= 1:  
    status = "FULL_MATCH"  
elif visible < expected:  
    status = "PARTIAL_OCCLUDED"  
else:  
    status = "OVERCOUNT"
```

5. Pipeline versions

5.1 Version overview

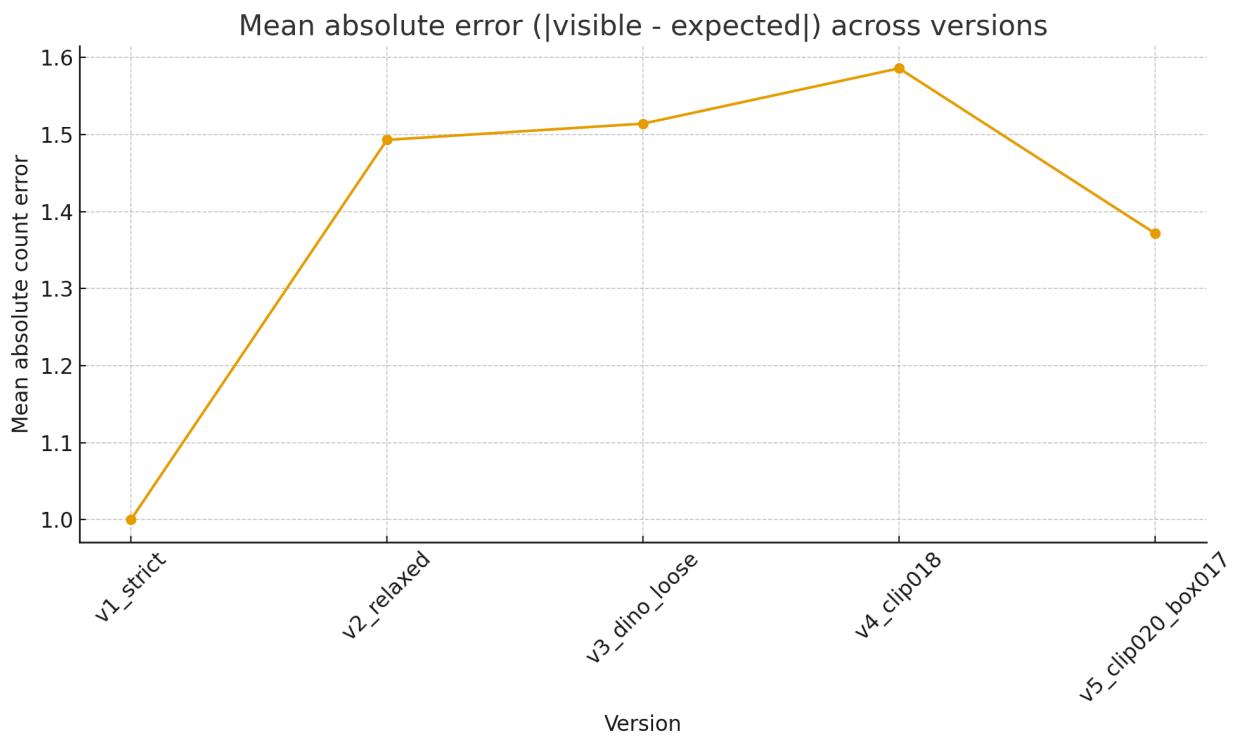
The main versions and their hyper-parameters:

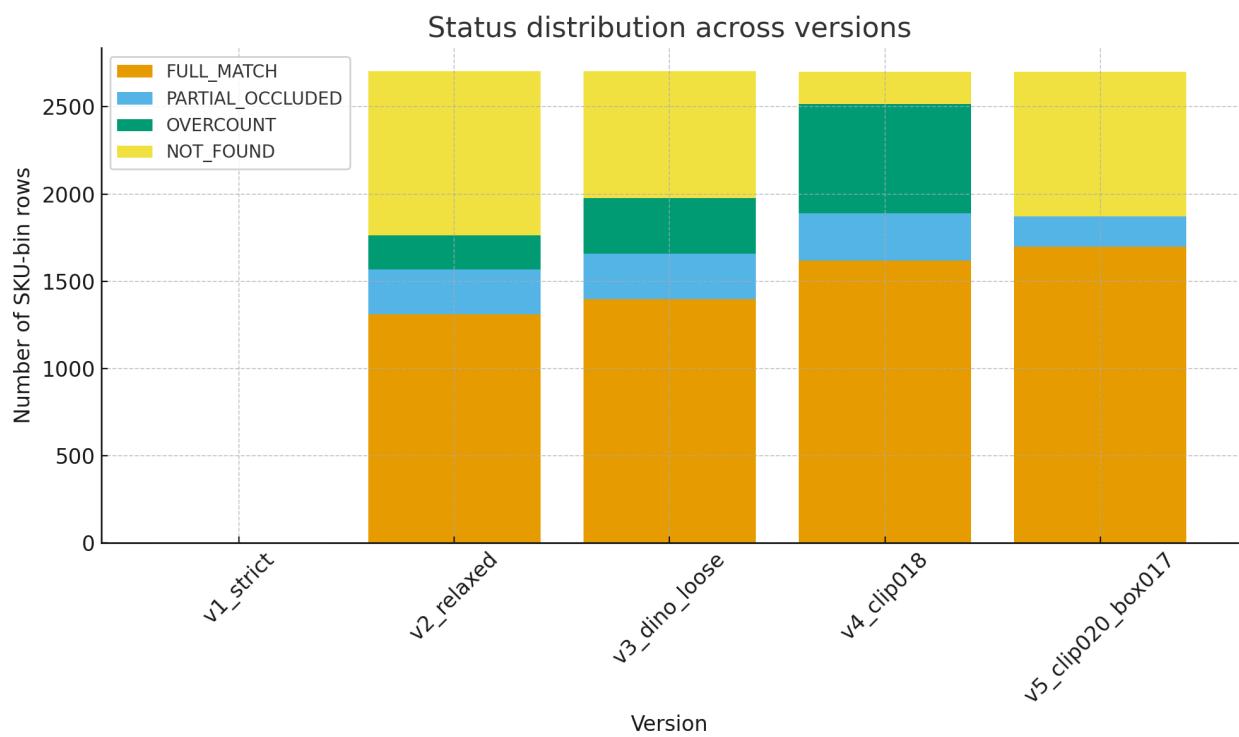
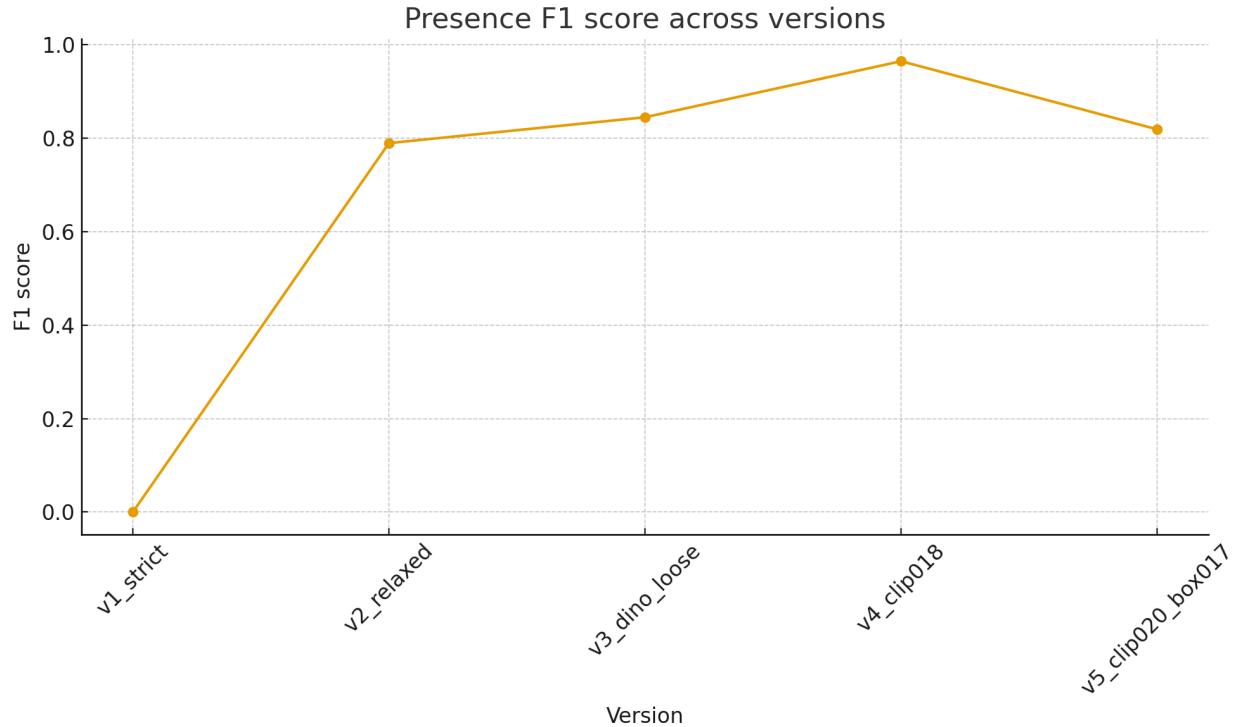
Version file	Label (infor mal)	BOX_THRES HOLD	TEXT_THRES HOLD	CLIP_THRES HOLD	Notes
validate_full_pipeline.py	v1 - strict	0.18	0.18	0.22	Very conservative, only run on 1 bin for sanity check
validate_full_pipeline_v2.py	v2 - baseline	0.18	0.18	0.20	Slightly relaxed CLIP; full dataset
validate_full_pipeline_v3_dino_loose.py	v3 - loose DINO	0.15	0.15	0.20	More permissive detector → more boxes
validate_full_pipeline_v4_clip018.py	v4 - loose CLIP	0.15	0.15	0.18	Very permissive CLIP → many more candidates
validate_full_pipeline_v5.py	v5 - final count-aware	0.17	0.18	0.20	Midpoint DINO, stricter CLIP, hard cap per SKU

Key design evolution:

- v1 → v2: relax CLIP from 0.22 to 0.20, keeping DINO thresholds; this was the first usable baseline.
- v2 → v3: loosen GroundingDINO (0.18 → 0.15) to increase recall; CLIP fixed at 0.20.

- **v3 → v4:** further loosen CLIP to 0.18 to maximise recall, at the cost of more false positives.
- **v4 → v5:**
 - Tighten DINO slightly to 0.17.
 - Restore CLIP to 0.20.
 - Introduce a **count-aware cap**: for each SKU, keep at most `expected + COUNT_TOLERANCE` boxes (with `COUNT_TOLERANCE = 1`).
 - Use the `status_from_counts` helper with ± 1 tolerance baked into `FULL_MATCH`.





6. Quantitative comparison of versions

Below we compare v2–v5 on the full dataset. v1 is omitted from tables because it only contains **3 rows / 1 bin** (debug run) and is not representative.

6.1 Status distributions

For each summary we count how many SKU rows fall into each status:

Summary file	FULL_MATCH	PARTIAL_OCCLUDE D	OVERCOUNT	NOT_FOUND
v2_relaxed	1311 (48.5%)	256 (9.5%)	196 (7.2%)	941 (34.8%)
v3_dino_loose	1396 (51.6%)	262 (9.7%)	319 (11.8%)	727 (26.9%)
v4_clip018	1617 (59.8%)	271 (10.0%)	629 (23.3%)	185 (6.8%)
v5_clip020_box0 17	1699 (62.9%)	174 (6.4%)	0 (0.0%)	829 (30.7%)

Observations:

- Moving from **v2 → v3 → v4**:
 - **FULL_MATCH** steadily increases (48.5% → 51.6% → 59.8%).
 - **NOT_FOUND** dramatically drops (34.8% → 26.9% → 6.8%) – recall is much higher.
 - But **OVERCOUNT** explodes (7.2% → 11.8% → 23.3%) as we relax thresholds.
- **v5**:
 - Achieves the **highest FULL_MATCH rate (62.9%)**.
 - Completely removes **OVERCOUNT** thanks to the count cap (**expected + 1**).
 - However **NOT_FOUND** climbs back to ~30.7% because we tightened detection / CLIP.

6.2 Detection recall and count error

Define:

- **Detection recall**: proportion of SKU rows with **visible > 0** (i.e. model detected at least one instance).
- **Mean absolute error (MAE)**: average of **abs(visible - expected)** across all rows.

Summary:

SF	R	B	DR								
v2_relaxed	2704	988	0.652	0.485	0.348	0.095	0.072	1.49	0.789		
v3_dino_loose	2704	988	0.731	0.516	0.269	0.097	0.118	1.51	0.845		
v4_clip018	2702	987	0.932	0.598	0.068	0.100	0.233	1.59	0.965		
v5_clip020_box017	2702	987	0.693	0.629	0.307	0.064	0.000	1.37	0.819		

Key trade-offs:

1. **v2 → v3** (looser DINO)
 - Detection recall increases (65% → 73%).
 - FULL_MATCH improves modestly (48.5% → 51.6%).
 - NOT_FOUND falls by ~8 percentage points.
 - OVERCOUNT rises from 7% to 12%.
 - MAE slightly worsens (1.49 → 1.51).
2. Interpretation: looser DINO helps find more SKUs but also introduces more false boxes, especially when prompts are generic or products are visually similar.
3. **v3 → v4** (looser CLIP)
 - Detection recall jumps to **93.2%** – almost every SKU has at least one detection.
 - NOT_FOUND becomes very rare (~7%).
 - FULL_MATCH improves further to ~60%.
 - However OVERCOUNT balloons to **23.3%**, nearly a quarter of all rows.
 - MAE is the worst (1.59).

4. Interpretation: CLIP at 0.18 allows many borderline matches; the model “sees” almost everything but over-counts aggressively, often counting background clutter as additional units.

5. **v4 → v5** (count-aware, mid thresholds)
 - DINO threshold raised to 0.17; CLIP back to 0.20; and a **count cap** of `expected + 1`.
 - Detection recall drops relative to v4 (93% → 69%), roughly back to between v2 and v3.
 - FULL_MATCH climbs to **62.9%**, the best among all versions.
 - PARTIAL_OCCLUDED shrinks to **6.4%**, lowest across versions.
 - OVERCOUNT becomes **0% by design**: the cap prevents heavy over-counting.
 - MAE improves from 1.59 (v4) to **1.37**, the best among v2–v5.

6. Interpretation: v5 intentionally sacrifices some recall to get much more reliable counts, avoiding scary overestimates. For an inventory/audit scenario, under-counting due to occlusion can be handled by human review, while over-counting can directly lead to wrong shipping or stock records.

6.3 Role of the count-aware cap

In v5, for each SKU the algorithm:

1. Sorts candidate boxes by CLIP similarity.
2. Limits accepted boxes to `min(num_candidates, expected + 1)`.

As a result:

- **Overcounting by more than 1** is impossible.
- In practice, the dataset has many cases where the model tends to over-fire slightly. The cap ensures those extra boxes are dropped.
- Because **FULL_MATCH** is defined with ±1 tolerance, a mild over-count (`expected=3, visible=4`) is still considered **FULL_MATCH**, which is acceptable for inventory scenarios.

This change is the main reason why **OVERCOUNT goes from 23.3% (v4) to 0% (v5)** even though the CLIP threshold is only slightly stricter.

7. Streamlit app: “Smart Bin Order Validator”

The `streamlit_app.py` provides an interactive demo of the pipeline:

7.1 Pipeline selection

The app defines a `PIPELINES` dictionary:

```
PIPELINES = {  
    "v2 - BOX=0.18, CLIP=0.20 (baseline)"  
    "v4 - BOX=0.18, CLIP=0.18 (relaxed CLIP)"  
    "v5 - BOX=0.17, CLIP=0.20 (count-aware)"}
```

In the `sidebar`, the user can:

- Choose the pipeline version (`v2`, `v4`, or `v5`).
- Choose which **bin ID** (image) to run on.
- Click a button to run the selected model.

The app uses cached functions:

- `@st.cache_resource` to load models (GroundingDINO + CLIP) per version.
- `@st.cache_data` to cache metadata / summary results.

7.2 Bin-level view

Once a bin is processed:

- The app shows the original **bin image** (with or without drawn boxes, depending on implementation).
- It presents a table (DataFrame) with columns like:
 - `asin`, `sku_name`, `expected`, `visible`, `status`

This is basically a visual front-end over the `summary_v*.csv` outputs for a single bin.

7.3 Order / invoice validation

The more interesting part is the “invoice” layer:

1. Function `build_order_from_bin(df_bin: pd.DataFrame)`:
 - Extracts unique SKUs for that bin with `expected` and `visible`.
 - For each row, the UI shows:

- Product name + ASIN
- A `number_input("Order_qty")` where the user specifies how many units they want to **order**.

Builds an order table:

```
asin, sku_name, order_qty, expected, visible
```

2. Line-item validation:

For each line, a simple rule:

```
if visible >= order_qty:
    order_status = "PASS"
else:
    order_status = "FAIL"
```

- The app shows an order table with `order_status` and an **order-level summary**:
“Order-level summary: X/Y line items are PASS.”

This turns the bin verification pipeline into a **smart order validator**:

- You visually see what's in the bin.
- You “place” an order based on that bin.
- The model checks whether the bin actually contains enough stock to fulfil the order.

This nicely links the ML pipeline to a real business use-case (inventory picking / order verification).

8. Implementation details

8.1 Requirements

`requirements.txt` contains:

```
torch
torchvision
Pillow
numpy
pandas
opencv-python
tqdm
```

```
boto3
transformers
groundingdino
git+https://github.com/openai/CLIP.git
```

Additional packages used in EDA/app (not all in the base file yet):

- `matplotlib` – EDA plots
- `streamlit` – app
- (optional) `seaborn`, `jupyter` – for notebooks

8.2 Scripts

- `s3_downloader.py` – retrieve images/metadata from S3.
- `validate_full_pipeline_v*.py` – core per-bin pipelines.
- `evaluate_model*.py` / `evaluate_model_v3_counts.py` – compute metrics and comparisons from summary files.
- `plot_results*.py` – CLI scripts to plot status distributions and trends.
- `EDA.py` – dataset + output analysis, as described above.

The project is fully **script-driven**, which makes experiments reproducible: you can regenerate each summary CSV by re-running the corresponding `validate_full_pipeline_*` script.

9. Discussion and insights

9.1 What worked well

- **GroundingDINO + CLIP** is a powerful combination for SKU-level detection in noisy bins.
- Careful tuning of **BOX and CLIP thresholds** showed a clear trade-off:
 - Lower thresholds → higher recall but more overcounts.
 - Higher thresholds → fewer false positives but more missed items.
- Introducing a **count-aware cap** in v5 dramatically improved practical usability:
 - Eliminated overcounting.
 - Achieved the highest rate of `FULL_MATCH` (62.9%).
 - Lowered mean absolute count error to 1.37.

9.2 Remaining issues

- **Occlusion** is still a big challenge:

- Many **PARTIAL_OCCLUDED** cases come from items behind others, partially cut off, or obstructed by packaging.
- **Fine-grained product differences:**
 - Some SKUs differ only by minor text/colour on packaging; GroundingDINO + CLIP may struggle to distinguish them.
- The current evaluation does **not** explicitly track false positives on *completely unexpected SKUs*, because the summaries are keyed by metadata SKUs only. Extra SKUs not in metadata are currently visible only in visualisations and not quantified.

9.3 Possible future improvements

- Use **instance segmentation** or **depth cues** to better handle overlapping items.
-

10. Conclusion

This assignment delivers an end-to-end system that:

1. Ingests bin images and structured metadata.
2. Uses **GroundingDINO** and **CLIP** to detect and verify products.
3. Produces per-SKU counts and status labels summarised in CSVs.
4. Iteratively improves model performance across **five pipeline versions**, culminating in **v5**, which balances recall, accurate counting, and safety against overcounting.
5. Provides both:
 - An **EDA toolkit** for understanding dataset and outputs.
 - A **Streamlit “Smart Bin Order Validator” app** that turns model outputs into a practical decision tool for checking whether a bin can fulfil an order.

In terms of comparison, **v4** achieves the highest raw recall but at the cost of many overcounts and larger count errors, while **v5** offers the best overall trade-off for real-world inventory.