Manga-pages to panel

Samip Shrestha

November 2025

Abstract

This document describes a system for automatically detecting comic/manga panels from page images. The approach uses computer vision techniques to find panel borders by analyzing dark content regions and their edges. It uses edge-based detection with gap tolerance to handle different comic layouts, and includes fallback logic when detection quality is low.

Contents

1 Introduction

Comic panel segmentation is the process of automatically identifying and extracting individual panels from a comic or manga page. This is useful for creating audio narrations, videos, or making comics more accessible.

1.1 Challenges

Detecting panels is tricky because:

- Panels have different shapes and sizes
- Gutters (space between panels) vary in width
- Some panels overlap or nest inside others
- Text and artwork can make borders unclear
- Comics read left-to-right, manga reads right-to-left

1.2 Approach

This system uses an edge-based method with:

- Edge detection to find panel borders
- 20% gap tolerance for incomplete borders
- Coverage checking to ensure quality results
- Reading order sorting for proper panel sequence

2 How It Works

The system processes a comic page through 6 steps. We'll use two real examples:

- **Page p015**: Normal case with 6 panels detected (784×1145 pixels)
- **Page p008**: Low coverage case triggering full-page fallback (784×1145 pixels)

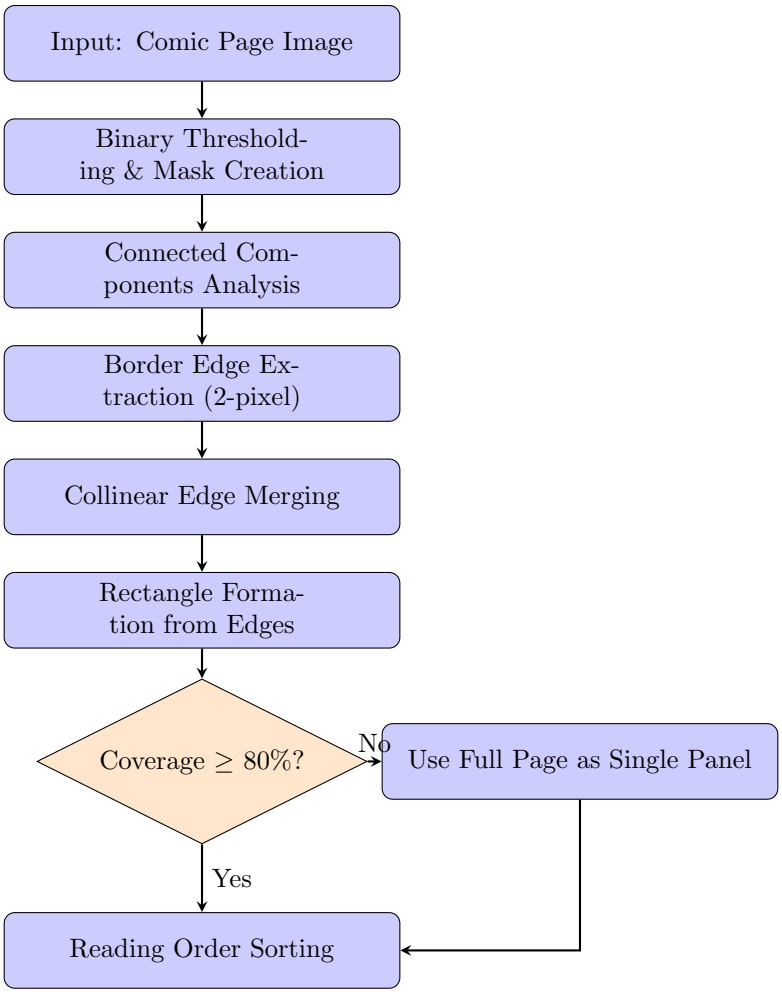

Figure 1: Comic Panel Segmentation Pipeline

2.1 Step 1: Create Binary Mask

What it does: Converts the image to grayscale and creates a binary mask where dark pixels (artwork, text) become white, and light pixels (gutters) become black. Uses a threshold of 50 on a 0-255 scale.

Why it's needed: Comic panels are separated by white space (gutters). By turning the image into just black and white, we can easily distinguish between "content" (the dark artwork) and "empty space" (the light gutters between panels). This makes it much easier to find where panels begin and end.

Mathematical formulation: Given a grayscale image $G(x, y)$ where pixel intensities range from 0 (black) to 255 (white), we apply inverse binary thresholding:

$$M(x,y) = \begin{cases} 255 & \text{if } G(x,y) < \theta \\ 0 & \text{if } G(x,y) \geq \theta \end{cases}$$

where $\theta = 50$ is our threshold value. This creates mask $M(x, y)$ where content pixels (dark) are white (255) and gutter pixels (light) are black (0).

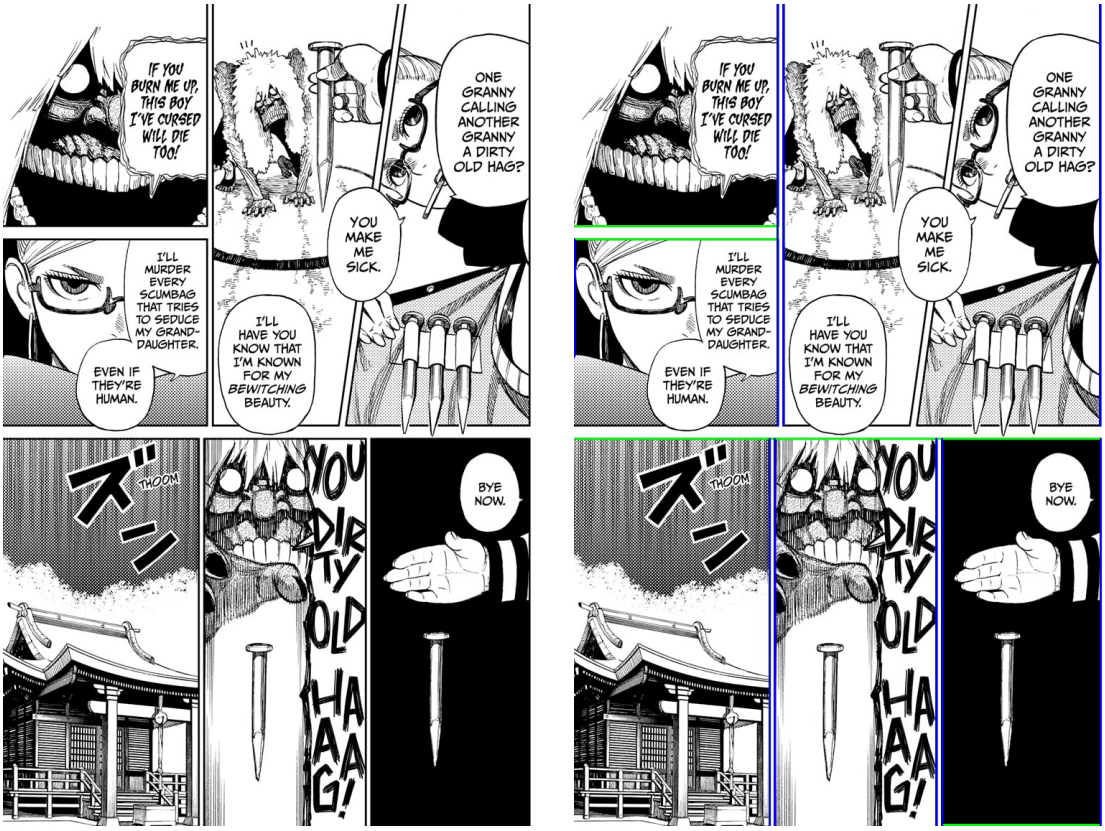

Figure 2: Left: Original page. Right: Detected border edges (red lines show panel boundaries)

```python
# Convert to grayscale
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
height, width = gray.shape

# Create binary mask (dark areas become white)
_, black_mask = cv2.threshold(
    gray,
    self.black_threshold,  # theta = 50
    255,
    cv2.THRESH_BINARY_INV
)
```

2.2 Step 2: Find Connected Components

What it does: Identifies all continuous dark regions (content areas) in the mask using connected component analysis. Each region represents artwork, text, or other panel content.

Why it's needed: This is the *critical foundation* that enables the entire edge-based detection approach. Without connected components analysis, edge detection would capture edges of everything in the image - character outlines, speech bubble borders, background details, and panel boundaries all mixed together, creating thousands of unusable false positives.

By first segmenting the image into 2,500+ individual content blobs, we can then extract edges *per blob*. When we find that many different blobs share the same edge position (e.g., 20 different content regions all end at $x = 300$), that common edge indicates a panel boundary rather than just content details. This per-blob analysis transforms noisy whole-image edges into clean panel boundaries.

Mathematical formulation: We use 8-connectivity to label connected components. Two pixels $p_1(x_1, y_1)$ and $p_2(x_2, y_2)$ are 8-connected if:

$$\max(|x_1 - x_2|, |y_1 - y_2|) = 1 \text{ and } M(x_1, y_1) = M(x_2, y_2) = 255$$

This partitions the mask into N disjoint regions $\{R_1, R_2, \ldots, R_N\}$ where each R_i represents a connected content blob.

Figure 3: Connected components colored uniquely (6,050 components found)

Example outputs:

- Page p015: 6,050 content patches found
- Page p008: 5,459 content patches found (more complex artwork)

2.3 Step 3: Extract Border Edges

What it does: For each content region found in Step 2, extracts the edges by finding the topmost, bottommost, leftmost, and rightmost 2-pixel borders. Only edges that are long enough (15% of width for horizontal, 10% of height for vertical) and continuous enough (80%) are kept.

Why it's needed: This step leverages the connected components from Step 2 to distinguish panel boundaries from content edges. By processing each of the 2,500+ content blobs individually, we extract only their outer boundaries. When multiple blobs share a common edge position, that indicates a panel border. This per-blob extraction is what prevents character outlines and internal artwork edges from being detected - we only see where content regions end, not the details within them. The 2-pixel thickness and continuity filters further refine these boundaries to identify true panel borders versus noise.

Mathematical formulation: For each component R_i, extract 2-pixel borders:

$$\text{Top}(R_i) = \{(x, y) : y \in [\min_y(R_i), \min_y(R_i) + 2]\}$$

$$\text{Bottom}(R_i) = \{(x, y) : y \in [\max_y(R_i) - 2, \max_y(R_i)]\}$$

Similarly for left and right edges. An edge E is valid if:

$$\text{length}(E) \geq \begin{cases} 0.15 \cdot W & \text{for horizontal edges} \\ 0.10 \cdot H & \text{for vertical edges} \end{cases}$$

$$\text{continuity}(E) = \frac{\#\text{ connected pixels}}{\text{total length}} \geq 0.80$$

where W and H are page width and height.

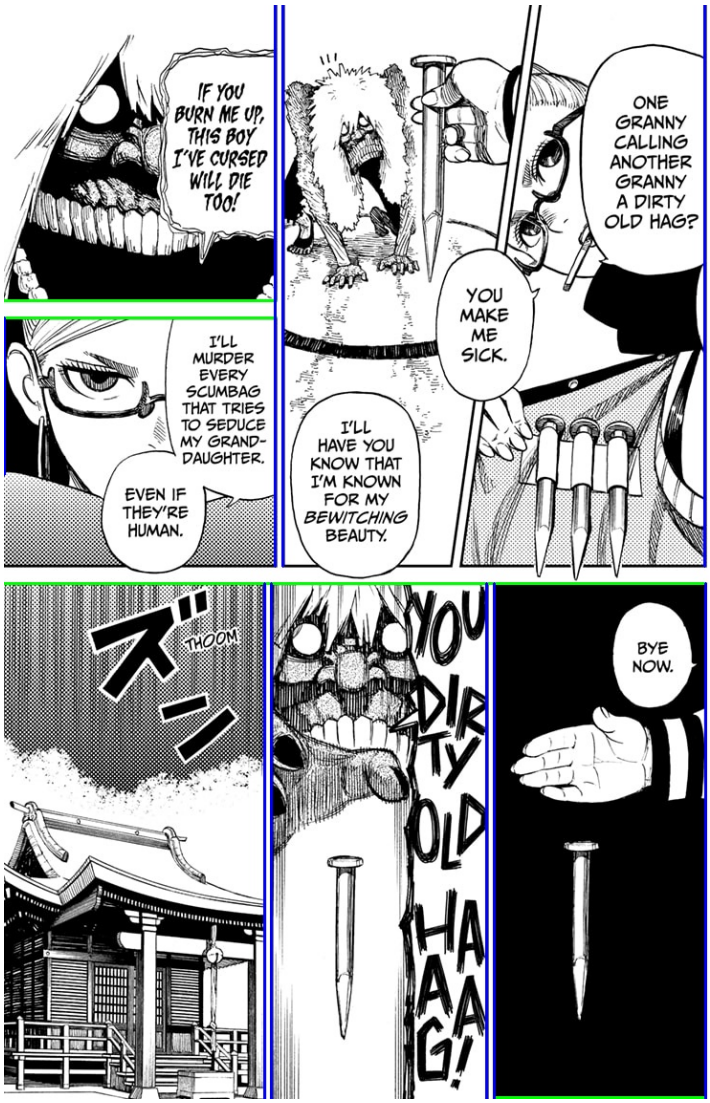

Figure 4: Border edges extracted from components (red lines show detected edges)

Example outputs:

- Page p015: 4 horizontal edges, 9 vertical edges
- Page p008: 3 horizontal edges, 3 vertical edges (fewer panel borders)

```
def _extract_border_edges(self, labels, num_labels, width, height):
    """Extract 2-pixel edges from connected components."""
    min_h_length = int(width * 0.15)   # 15% of width
    min_v_length = int(height * 0.10)  # 10% of height

    # Initialize edge images
    horizontal_borders = np.zeros((height, width), dtype=np.uint8)
    vertical_borders = np.zeros((height, width), dtype=np.uint8)

    for label in range(1, num_labels):
        patch_mask = (labels == label).astype(np.uint8) * 255

        # Extract top and bottom edges (2-pixel strips)
```

```
        rows_with_white = np.where(np.any(patch_mask == 255, axis=1))[0]
        if len(rows_with_white) > 0:
            top_row = rows_with_white[0]
            bottom_row = rows_with_white[-1]

            # Process top edge
            top_edge = patch_mask[top_row:min(top_row+2, height), :]
            # [validation and storage logic]

            # Process bottom edge
            bottom_edge = patch_mask[max(bottom_row-1, 0):bottom_row+1, :]
            # [validation and storage logic]

        # Similar processing for left and right edges
        # ...

    return h_edges, v_edges
```

2.4 Step 4: Merge Collinear Edges

What it does: Edges from different content regions that are on the same line (within 5 pixels) and close together (within 30 pixels) are merged into continuous edges.

Why it's needed: A single panel border often gets detected as multiple small edge segments because different content blobs (a character here, a speech bubble there) touch the same border. For example, if a panel has three speech bubbles along the top, we'd detect three separate top edges. Merging them gives us one continuous top edge, which is what we need to form a complete panel rectangle.

Mathematical formulation: Two horizontal edges e_1 at y_1 and e_2 at y_2 are merged if:

$$|y_1 - y_2| \leq \tau \quad \text{and} \quad \text{overlap}(e_1, e_2) > 0$$

where $\tau = 30$ pixels is the tolerance. The merged edge position is:

$$y_{\text{merged}} = \frac{\text{length}(e_1) \cdot y_1 + \text{length}(e_2) \cdot y_2}{\text{length}(e_1) + \text{length}(e_2)}$$

Similar logic applies to vertical edges using x-coordinates.

Example outputs:

- Page p015: Merged to 4 horizontal + 7 vertical edges
- Page p008: Merged to 3 horizontal + 2 vertical edges

2.5 Step 5: Form Rectangles from Edges

What it does: Tries all combinations of horizontal and vertical edges to form rectangular panels. A rectangle is valid if: (1) it's big enough (at least 10% of page dimensions), and (2) the missing border pieces add up to less than 20% of the perimeter. Rectangles with corner connections are preferred. Smaller panels are chosen first to avoid overlap.

Why it's needed: We have a bunch of horizontal and vertical lines - now we need to figure out which ones form actual panels. We test every possible rectangle we can make from these lines. Not all rectangles are real panels though - some are too small (just a speech bubble), and some have too much missing border (edges that don't fully connect). We allow up to 20% missing border because sometimes panel borders fade into artwork or have artistic gaps.

Mathematical formulation: Given horizontal edges $\mathcal{H} = \{h_1, h_2, \ldots, h_m\}$ and vertical edges $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$, form candidate rectangles $R = (x_1, y_1, w, h)$ from all combinations where $y_1, y_2 \in \mathcal{H}$ and $x_1, x_2 \in \mathcal{V}$.

A rectangle is valid if:

1. **Size constraint:** $w \geq 0.10 \cdot W$ and $h \geq 0.10 \cdot H$
2. **Border completeness:**
$$\text{gap\_ratio} = \frac{\text{missing\_border\_length}}{2(w+h)} \leq 0.20$$
3. **Corner bonus:** Rectangles with corners formed by edge intersections are preferred

Example output for page p015: 6 panels detected

- Panel 1: [291, 0, 441, 605] - Large right panel
- Panel 2: [0, 0, 291, 309] - Top left
- Panel 3: [0, 327, 291, 278] - Middle left
- Panel 4: [512, 605, 220, 538] - Bottom right
- Panel 5: [279, 605, 225, 538] - Bottom center
- Panel 6: [0, 605, 272, 538] - Bottom left

2.6 Step 6: Coverage Check and Reading Order

What it does: Checks if the detected panels cover at least 80% of the page. If not, uses the whole page as a single panel. Finally, sorts panels in reading order - left-to-right for comics, right-to-left for manga.

Why it's needed: If our detected panels only cover 50% of the page, something went wrong - we're missing major content. This usually happens with artistic pages, title pages, or full-page spreads where there aren't clear panel borders. Rather than give bad results, we admit defeat and treat the whole page as one panel. The reading order is crucial for creating narrations or videos - readers need to experience panels in the correct sequence.

Mathematical formulation: Given detected panels $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$, compute coverage:

$$\text{coverage} = \frac{\text{area}\left(\bigcup_{i=1}^{k} P_i\right)}{W \times H}$$

If coverage < 0.80, fallback to $\mathcal{P} = \{(0, 0, W, H)\}$.

For reading order, sort panels by:

$$\text{sort\_key}(P_i) = \begin{cases} -x_i + \epsilon \cdot y_i & \text{manga (right-to-left)} \\ x_i + \epsilon \cdot y_i & \text{comic (left-to-right)} \end{cases}$$

where $\epsilon = 0.001$ breaks ties by vertical position.

Example - Page p015 (Normal):

- 6 panels detected with good coverage
- Panels sorted by reading order (manga: right-to-left)
- Final output: 6 individual panels

Example - Page p008 (Low Coverage):

- Initial detection found panels covering <80% of page
- Triggered fallback to full-page mode
- Final output: 1 panel = entire page [0, 0, 784, 1145]

3 Example Results

3.1 Example 1: Dandadan Chapter 3, Page 15 (Normal Case)

- Page size: 784 × 1145 pixels
- Content patches: 6,050
- Edges detected: 4 horizontal, 9 vertical
- Detected: 6 panels
- Coverage: >80% (passed)

Table 1: Detected Panels - Page 15

Panel	X	Y	Width	Height	Area
1	291	0	441	605	266,805 px
2	0	0	291	309	89,919 px
3	0	327	291	278	80,898 px
4	512	605	220	538	118,360 px
5	279	605	225	538	121,050 px
6	0	605	272	538	146,336 px

This page shows the typical case: clear panel borders were detected, merged properly, and formed valid rectangles with good coverage.

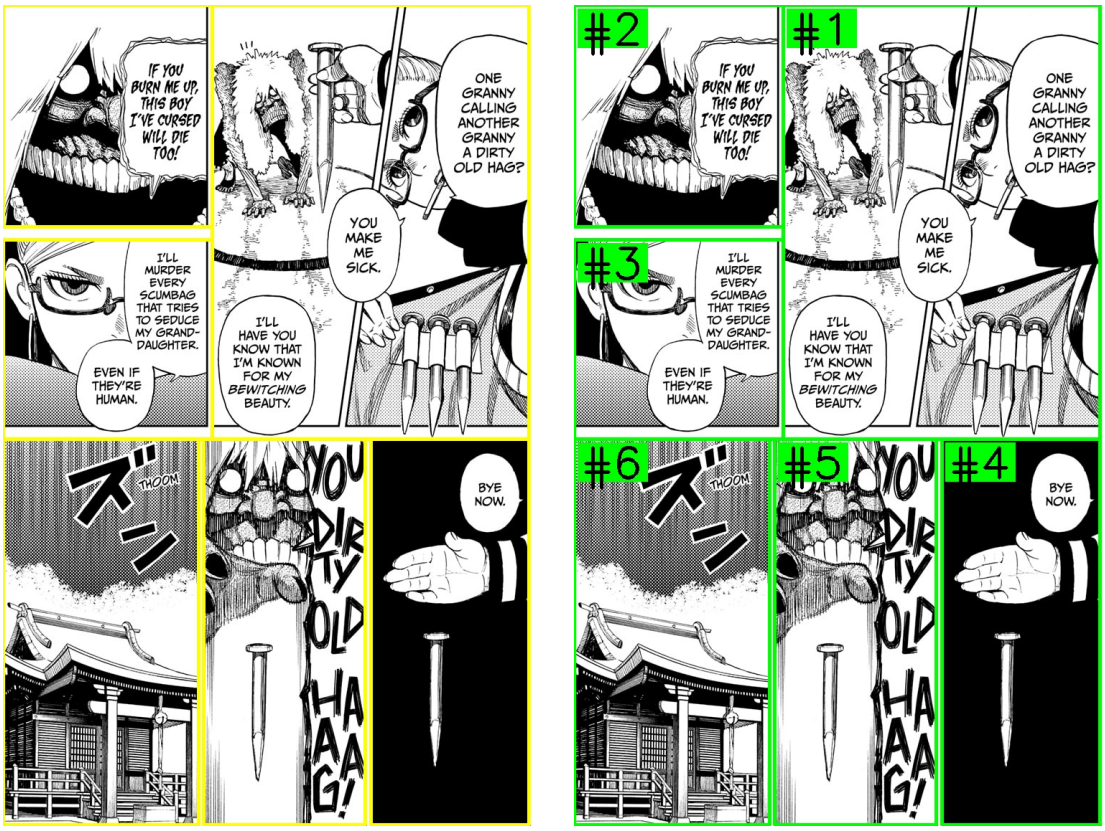

Figure 5: Left: Detected panel boundaries. Right: Final panels numbered in reading order

3.2 Example 2: Dandadan Chapter 3, Page 8 (Fallback Case)

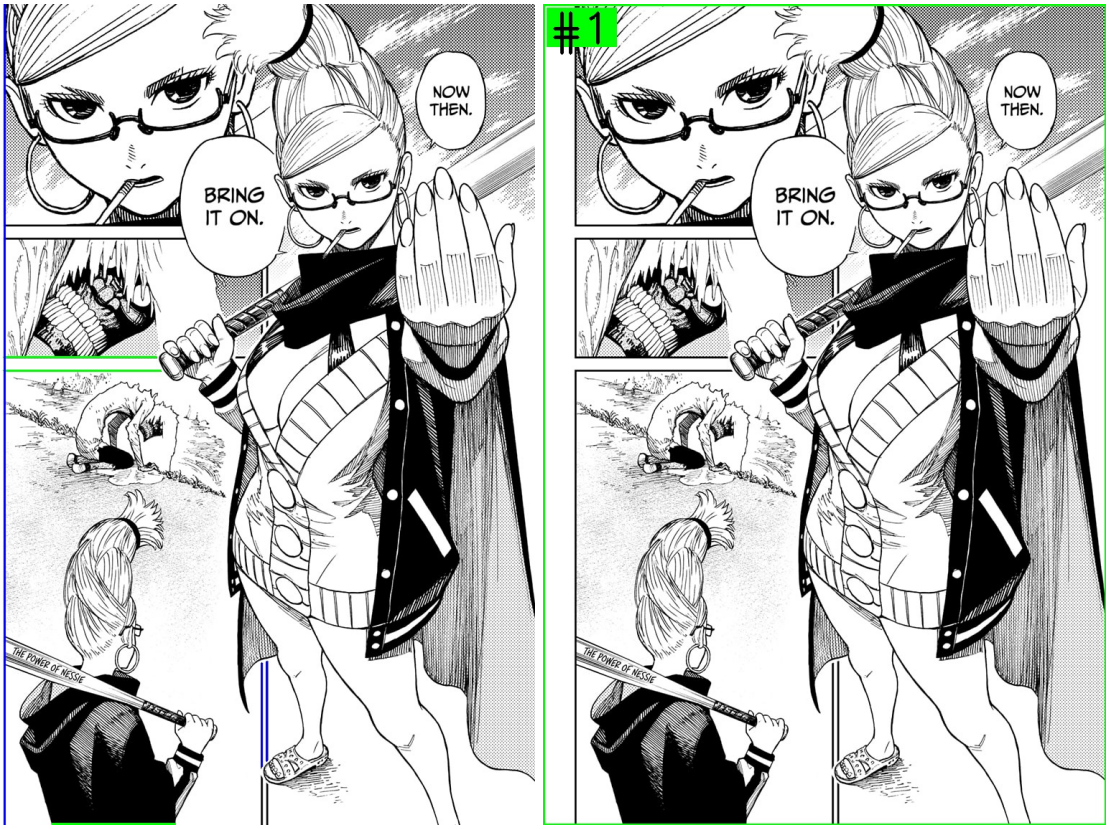

Figure 6: Left: Detected panel boundaries (insufficient coverage). Right: Fallback to full-page

- Page size: 784 × 1145 pixels
- Content patches: 5,459
- Edges detected: 3 horizontal, 3 vertical
- Initial detection: Panels with low coverage (<80%)
- Result: Fallback to full page

Table 2: Detected Panels - Page 8

Panel	X	Y	Width	Height	Area
1	0	0	784	1145	897,680 px

This page triggered the fallback mechanism. Despite detecting edges, the resulting panels didn't cover enough of the page (likely due to complex artwork or unclear borders), so the system used the entire page as a single panel.

4 Key Parameters

Table 3: Tunable Settings

Parameter	Value	What it does
Black threshold	50	Pixels darker than this are considered content
Merge tolerance	30 px	Max gap to merge edge segments
Max gap %	20%	Max missing border allowed for panels
Min edge length	15% W / 10% H	Minimum edge size to consider
Min panel size	10% W / H	Minimum panel dimensions
Coverage threshold	80%	Minimum page coverage needed

5 Issues Fixed During Development

5.1 Edge Fragmentation

Initially, edges were extracted separately for each content region, creating many small fragments instead of continuous lines. Fixed by drawing all edges onto shared border images and consolidating them.

5.2 Duplicate Panels

Full-page frames were being detected as separate panels on top of the actual panels. Added logic to remove "container" panels that mostly just contain other panels.

5.3 Low Coverage

Some pages only had 52% coverage, missing large parts of the page. Added an 80% threshold check - if coverage is too low, use the whole page as one panel.

5.4 Missing Panels in Output

Initially thought panels were being missed, but verification showed extraction was working fine - the panels were being filtered out by downstream AI analysis based on importance scores.

6 Limitations and Edge Cases

6.1 What Works

- Full-page spreads (detected via low coverage)
- Variable gutter widths
- Different reading orders (manga vs comics)
- Nested or overlapping panels

6.2 What Doesn't Work Well

- Panels without clear borders may merge together
- Very complex or artistic layouts may trigger full-page fallback
- Dark backgrounds can confuse the threshold-based detection

7 Implementation

7.1 Technologies Used

- Python 3.8+
- OpenCV for image processing
- NumPy for array operations

7.2 Usage

```python
from comic_processor.utils.panel_extractor import PanelExtractor

extractor = PanelExtractor()
panels = extractor.extract_panels(
    image_path="page_001.jpg",
    content_type="comic"  # or "manga"
)

for panel in panels:
    print(f"Panel {panel['panel_number']}")
    print(f"  Position: {panel['bbox']}")
    print(f"  Area: {panel['area']} px")
```

7.3 Output Format

Each panel is a dictionary with:

- `panel_number`: Panel sequence number
- `bbox`: [x, y, width, height]
- `image`: Cropped panel image
- `area`: Panel area in pixels

A `metadata.json` file is also saved with panel information.

8 Conclusion

This system provides a simple, fast way to detect comic panels using computer vision. It doesn't require machine learning or training data, works on different comic styles, and handles edge cases through fallback strategies.

The edge-based approach with gap tolerance (up to 20% missing border allowed) and coverage checking makes it robust enough for real-world use in comic analysis pipelines.