ChatGPT

Improving Text Overlay in Nanobanana MCP Image Editing

Issues with Current Text Overlay

The existing text overlay functionality in the Nanobanana MCP image workflow has several problems: - **Misalignment:** Text labels (e.g. in speech bubbles) are not positioned as intended. They may not be centered within the bubble or may overlap important image content. - **Clipping of Text:** Longer text strings sometimes get cut off or only partially rendered. Portions of letters (like descenders) or entire words may not appear. - **Inconsistent/Unclear Font Rendering:** The chosen font appears too small, jagged, or stylistically off. In some cases the text is hard to read or doesn't match the intended style.

These issues make it difficult for the user to create stylized images with embedded captions. Below we outline improvements to address each problem, ensuring captions are **readable**, **properly placed**, and **fully rendered**.

Automatic Text Box Sizing and Wrapping

One root cause of clipping and misalignment is the lack of dynamic text layout. Currently, the overlay might be drawing text in a fixed-size area or single line, causing overflow. **The tool should automatically size the text box and wrap text** as needed: - **Word Wrapping:** Instead of allowing long strings to run off the image or out of a bubble, break text into multiple lines that fit within defined width boundaries. Pillow (PIL) does not automatically wrap text – there's no built-in function to do so [1]. We need to implement this manually. For example, we can use Python's `textwrap` utility to split a paragraph into lines of a target width (in characters or pixels) and then draw each line separately [2] [3]. This ensures longer captions are neatly broken into lines. - **Dynamic Box Sizing:** The speech bubble or text background shape should expand to accommodate the text. After wrapping the text, compute the total text block dimensions (width and height). We can do this by summing up the width/height of each line using Pillow's text size measurement. For instance, `draw.textsize()` or `font.getsize()` gives the pixel size of a text string for a given font [4] [3]. Using such measurements, adjust the bubble's size or the coordinates so that the entire text fits with some padding. This prevents any text from being cut off at the edges. - **Pixel-Accurate Layout:** A simple approach might wrap by character count (e.g. 40 chars per line) as a rough proxy [2], but for accuracy with variable-width fonts, use pixel measurements. A more robust algorithm is to accumulate words and check the rendered width each time. For example, one can iterate over words and build lines until adding another word would exceed the max width (using `font.getlength()` or `font.getsize()` to measure) [5]. This way, the wrapping respects the actual font metrics, avoiding lines that are too long. The end result can be assembled into a multi-line string or drawn line by line.

By implementing smarter text wrapping and auto-sizing, the MCP tool will **prevent text overflow** and ensure the speech bubble or label background always matches the text content size.

Centered and Aligned Text Placement

To address text misalignment, we need to **properly center or align text within its container** (such as a speech bubble shape or designated label area): - **Horizontal & Vertical Centering:** By default, Pillow's `ImageDraw.text` treats the given `(x, y)` as the top-left corner of the text bounding box [6]. This means if you draw text at the bubble's top-left, it won't be centered. To center text, calculate an appropriate offset. A common method is to measure the text and offset by half its dimensions. For example, if the bubble rectangle's center is `(center_x, center_y)`, and the text width/height is `(w, h)`, draw at `(center_x - w/2, center_y - h/2)` [4]. This will place the text so it's exactly centered in both directions. - **Using Pillow's Anchor Feature:** Modern versions of Pillow (8.0.0 and above) provide a convenient `anchor` parameter for centering text. You can specify an anchor mode like `"mm"` (middle x, middle y) so that the coordinate given is treated as the text's center [7]. For example: `draw.text((cx, cy), "Caption", font=my_font, anchor="mm")` will automatically center the text `"Caption"` at `(cx, cy)` [7]. This greatly simplifies vertical and horizontal centering, as Pillow handles the alignment internally. - **Alignment for Multiline Text:** If the caption spans multiple lines (after wrapping), you should also handle alignment of those lines. Pillow's `multiline_text()` will accept an `align` parameter to align each line left, centered, or right within the multiline text box [6]. For instance, if you want center-aligned lines, use `align="center"` when drawing the multiline text. Alternatively, when drawing lines one by one, you can individually position each line's x-coordinate so that they are centered under each other (or aligned to a margin). The example below demonstrates centering each line of wrapped text by computing each line's width and offsetting appropriately [3] [8].

By ensuring proper alignment, text will appear **neatly centered in speech bubbles** (or aligned as desired), rather than drifting to one side or overlapping the bubble's border or underlying image elements.

Padding and Margin Configuration

Even with correct centering, adding some **padding** around the text is important for a polished look. The tool should let users configure padding and margins: - **Inner Padding:** This is space between the text and the bubble or label outline. For example, a speech bubble should have a bit of blank space on all sides of the text so the text isn't touching the edges. Implementation-wise, after measuring the text block, you can add a few pixels (configurable) to width and height when drawing the bubble background. Also offset the text position by that padding amount. This ensures the text sits comfortably within the bubble. - **External Margin:** If the user wants the text box away from other image content, a margin could define how far the bubble is placed relative to other objects or image borders. This might be handled outside the low-level drawing (e.g. when deciding the bubble's position on the image), but exposing it as a parameter makes placement more flexible. - **Alignment Options:** In addition to centering, allow the user to specify if text should be left-justified or right-justified within the bubble. This could be useful for certain layouts or styles. As noted, Pillow's multiline text drawing can handle left/center/right alignment of lines automatically [6]. The tool's API could simply pass along an alignment setting. For vertical alignment (top, middle, bottom within the bubble), you can similarly calculate different y offsets or use anchor modes (e.g. `"ma"` for middle horizontally, ascender/top vertically, or `"md"` for middle + descender/bottom) depending on the effect. Generally, center-middle works for speech bubbles, but for labels at the top or bottom of an image, top or bottom alignment might be preferred – so it's good to allow it.

By making padding and alignment configurable, the MCP image tool will let users fine-tune the appearance of text labels, ensuring **no overlap or crowding** between text and other visual elements.

Font Selection and Fallback

The clarity issues with the text likely stem from the font being used. To improve consistency and readability: - **Use High-Quality TrueType Fonts:** Pillow can load TrueType or OpenType font files via `ImageFont.truetype`. Instead of relying on the default Pillow bitmap font (which is very small and plain), load a known good font. For example, using a font like **Noto Sans** or **Roboto** will yield clear, antialiased text. Noto Sans in particular is designed to be universal (covering many languages) and renders cleanly at various sizes. In fact, Pillow's own documentation often uses Noto Sans in examples [9]. Similarly, classic fonts like **Arial** are well-tested and render reliably. Ensure the font file is available and specify a reasonable font size for the image resolution. - **Font Size and Styling:** Provide options to adjust font size, weight, and style. The font size should perhaps auto-scale with image dimensions or user preference. For a speech bubble in a typical image, a moderate font size (e.g. 20–40 px) might be appropriate, but if the image is high-resolution, larger text might be needed. Allowing bold or italic variants (if available for the chosen font) can also help match the style of the image. - **Fallback Fonts:** In cases where the primary font doesn't support certain characters (for example, if the user adds emoji or non-Latin scripts), the tool should have a strategy so those characters aren't just missing or garbled. Pillow doesn't automatically fallback to another font if a glyph is missing [10]. One approach is to detect unsupported characters and manually substitute a secondary font for those. However, a simpler solution is to choose a font like Noto Sans from the start, which has extensive glyph coverage. If needed, the tool could maintain a small list of fallback fonts (e.g. try drawing with Font A, and if a character isn't renderable, switch that character or the whole text to Font B). This ensures the text always appears correctly. - **Consistent Rendering:** By explicitly loading a font and drawing text with it, we avoid platform-dependent font issues. This will eliminate the "broken or inconsistent" appearance by using a predictable typeface. It's also wise to **antialias** the text (Pillow does this by default for TrueType fonts) so it's smooth. If the text still looks too small or thin, consider using a heavier font weight or a slight outline/shadow to improve contrast against the background.

Overall, picking a solid font and handling font fallback will make the caption text **much clearer and more reliable** in appearance.

Preventing Text Clipping

Text getting cut off (either horizontally or vertically) is a known issue when rendering text with PIL/Pillow. This can happen if the drawing canvas is too small or due to how Pillow calculates text bounding boxes. To avoid any part of the text from being clipped: - **Adequate Canvas/Bubble Size:** As mentioned, compute the required size of the text box and ensure the image or region is slightly larger. If text is drawn at the very edge of an image, parts can be truncated. A simple fix is to always leave a margin or padding (even invisible) around drawn text. - **Adjust for Descenders:** Some characters like "g, p, y" have portions (descenders) that extend below the baseline and may get cut if the bounding box is tight. Similarly, accents on top of letters might get cut off at the top. Ensure the vertical padding accounts for these. In practice, you might add a few pixels extra above and below the text's measured height to be safe. - **Known Pillow Bug and Workaround:** Pillow has had a long-standing quirk where certain letters' edges can be clipped due to the font rendering internals [11]. One popular workaround is to append an extra **space** at the end of the text string when drawing. This dummy space often provides the extra buffer needed so that the last character isn't cut off [12]. For example, if drawing `"Hello!"`, actually draw `"Hello! "` (with a space at end). This trick can prevent the final character or descender from clipping. - **Dynamic Font Scaling:** If the user specifies a text string that is extremely long for the given image, another approach is to dynamically reduce the font size until it fits. The tool could detect that the text (or a wrapped block of text) still doesn't fit in the bubble or image and then iteratively shrink the font size. Conversely, it could also enlarge the speech bubble to fit the text. Decide which behavior makes sense

(probably resizing the bubble/container is preferable to shrinking text too much, since tiny text becomes illegible). In either case, ensure before drawing that the text will fit fully. This avoids drawing text and finding it's cut off. - **Alternate Rendering Backends:** In rare cases where Pillow's text rendering is insufficient, there are libraries like **AggDraw** or **Pycairo** that can render text with fine-grained control. AggDraw, for instance, is noted to handle text without the clipping issue [13]. Incorporating another library is likely not necessary if we implement the above measures, but it's something to keep in mind if high-quality text rendering becomes a priority.

By adding these safeguards, the tool will **eliminate clipped text**, ensuring that complete captions are visible in the final image (no half-rendered letters or missing bottoms of letters).

Speech Bubble Shape and Tail Drawing

Finally, to help users create comic-style speech bubbles with text, the tool should provide a helper to draw the bubble background itself (not just the text). This involves drawing a shape (ellipse or rounded rectangle) for the bubble and a **tail** pointing to the speaker or source of the speech: - **Bubble Background Shape:** A common approach is to use a rounded rectangle or an ellipse as the bubble. Pillow's `ImageDraw` now directly supports rounded rectangles (available since Pillow 8.2) [14]. For example, `draw.rounded_rectangle((x1, y1, x2, y2), radius=10, fill=..., outline=...)` will draw a rectangle with curved corners [15]. You can use the measured text dimensions plus padding to determine `(x1,y1,x2,y2)`. Alternatively, an ellipse (`draw.ellipse(...)`) can be used for a thought bubble or a more circular balloon. Ensure the fill color and outline (border) color match the desired cartoon style (often white fill with black outline, or any stylized colors). - **Drawing the Tail:** The speech bubble "tail" is typically a triangular or curved pointer. A simple implementation is to draw a triangle polygon. For example, if the bubble should point to the bottom-right (towards a character's mouth), you might take a point on the bubble's bottom edge and two points further down towards that direction to form a triangular tail. Pillow's `draw.polygon()` can draw a filled triangle given three coordinates. For instance: `draw.polygon([ (x2-10, y2), (x2+5, y2+20), (x2+15, y2) ], fill=..., outline=...)` would create a small triangular tail at the bottom of the bubble (this is just an illustrative coordinate example). The tool could provide parameters like `tail_side` (e.g. "bottom-left", "top-right", etc.) and `tail_offset` or `tail_length` to control where and how the tail is drawn. The tail should connect smoothly to the bubble – one might slightly overlap the shapes or use the same outline color so it looks like one continuous shape. - **Automation:** A helper function `draw_speech_bubble(text, position, **options)` could wrap all this: it would calculate the required bubble size from the text (applying the above auto-wrapping and padding), then draw the bubble shape with tail, and finally draw the text centered within it. Options could include colors, tail direction, padding, alignment, etc. By encapsulating this, the user can easily annotate images with speech bubbles by one function call. - **Layering Order:** When composing the image, draw the bubble **before** drawing the text (so the text appears on top of the bubble). Also, ensure the bubble (background) is drawn on the image (or on a separate image that is then composited) with proper transparency if needed. For example, if the bubble should have a semi-transparent look or if it overlaps the original image content, using RGBA mode and semi-transparent fills might be desirable.

With a dedicated speech bubble drawing utility, users can create expressive annotations in one step. The text will be **properly enclosed in an attractive bubble with a pointer**, enhancing the overall visual presentation. The combination of the above improvements – smarter text layout and a drawing helper – will result in stable and visually accurate rendering of captions on images, meeting the user's goal of stylized images with embedded text.

Summary

By implementing these changes, the Nanobanana MCP image editing tool will significantly improve how it handles text overlays. Text will be automatically wrapped and scaled to fit, drawn with clear fonts, centered perfectly in customizable speech bubbles, and free of clipping issues. These enhancements empower users to create annotated visuals (like comic-style speech balloons or labeled images) with confidence that the captions will appear **readable, well-aligned, and aesthetically pleasing**.

Sources:

- Pillow text wrapping and centering examples [3] [8]
- Pillow text alignment and anchor documentation [7] [6]
- Known PIL text clipping issue and solutions [12]
- Pillow usage of high-quality fonts (Noto Sans) [9]
- Pillow drawing shapes (rounded rectangle for bubbles) [16]

---

[1] Wrap and Render Multiline Text on Images Using Python's Pillow Library - DEV Community
https://dev.to/emiloju/wrap-and-render-multiline-text-on-images-using-pythons-pillow-library-2ppp

[2] [5] python - Wrap text in PIL - Stack Overflow
https://stackoverflow.com/questions/8257147/wrap-text-in-pil

[3] [4] [7] [8] python - Center-/middle-align text with PIL? - Stack Overflow
https://stackoverflow.com/questions/1970807/center-middle-align-text-with-pil

[6] Python PIL | ImageDraw.Draw.text() - GeeksforGeeks
https://www.geeksforgeeks.org/python/python-pil-imagedraw-draw-text/

[9] Text anchors - Pillow (PIL Fork) 11.3.0 documentation
https://pillow.readthedocs.io/en/stable/handbook/text-anchors.html

[10] python imaging library - How to setup fallback fonts in Pillow? - Stack Overflow
https://stackoverflow.com/questions/64651021/how-to-setup-fallback-fonts-in-pillow

[11] [12] [13] python - fonts clipping with PIL - Stack Overflow
https://stackoverflow.com/questions/1933766/fonts-clipping-with-pil

[14] [15] [16] Drawing Rectangles with Rounded Corners with Pillow and Python - Mouse Vs Python
https://www.blog.pythonlibrary.org/2021/05/12/drawing-rectangles-with-rounded-corners-with-pillow-and-python/