Simple Agents Outperform *Experts* in Biomedical Imaging Workflow Optimization

Supplementary Material

The sections of our supplementary material are organized as follows:

- Section A: Qualitative results (visualization of the expert and agent processed images and results).
- Section B: Agent function deployed to production (GitHub PR screenshot).
- Section C: Expert functions and agent generated functions.
- Section D: Git History Analysis of expert baseline functions.
- Section E: Analysis metric details.
- Section F: Additional dataset details.
- Section G: Prompt details.
- Section H: Non-Agentic AutoML Baseline.
- Section I: AIDE Baseline.
- Section K: MedSAM-XRay prospective validation.
- Section J: Additional robustness analysis (validation robustness, runs ablation, success rates).
- Section L: Computational requirements.
- Section M: API list.

A. Visualizations

A.1. Polaris

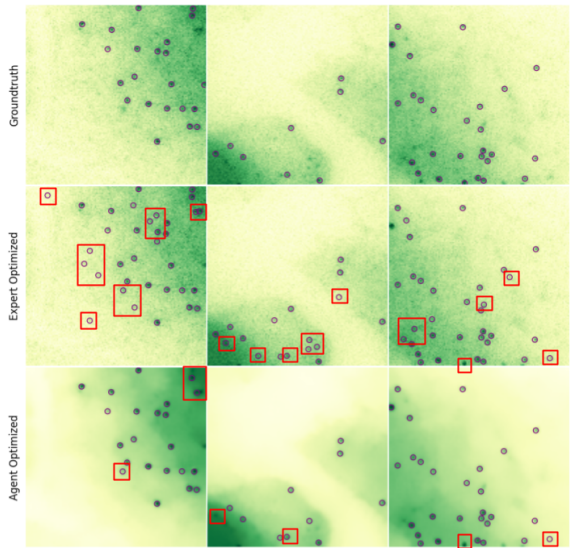


Figure 1. Example images showing (Top) Groundtruth, (Middle) Predictions using expert-optimized function pairs, and (Bottom) Prediction using agent-optimized function pairs. Prediction errors are marked with red boxes.

A.2. Cellpose

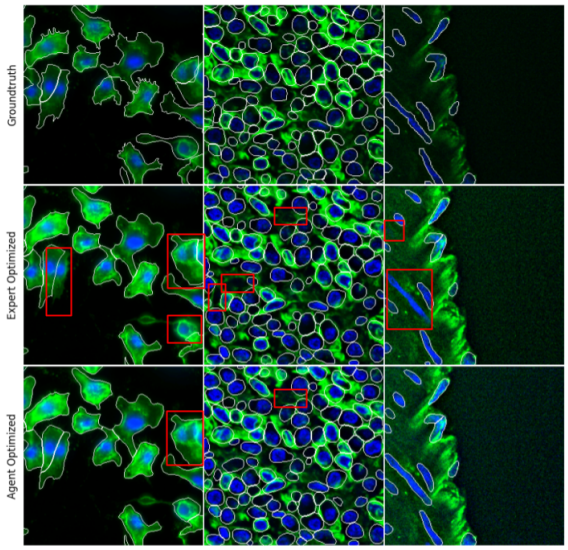


Figure 2. Example images showing (Top) Groundtruth, (Middle) Predictions using expert-optimized function pairs, and (Bottom) Prediction using agent-optimized function pairs. Prediction errors are marked with red boxes.

A.3. MedSAM

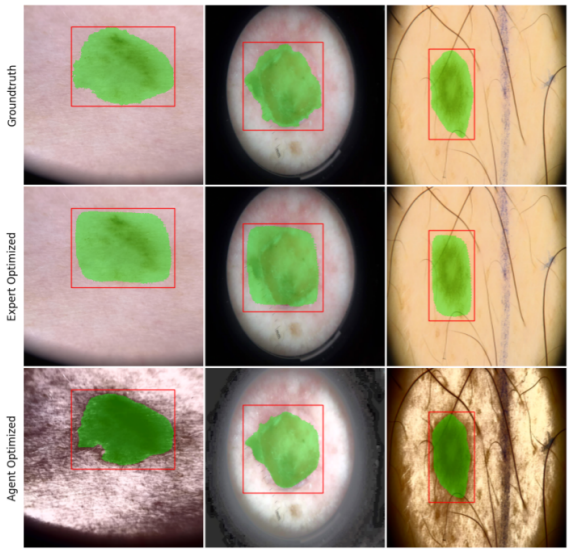


Figure 3. Example images showing (Top) Prompt box and groundtruth, (Middle) Prompt box and prediction using expert-optimized function pairs, and (Bottom) Prompt box and prediction using agent-optimized function pairs. Prediction errors are marked with red boxes.

B. Agent function deployed to production

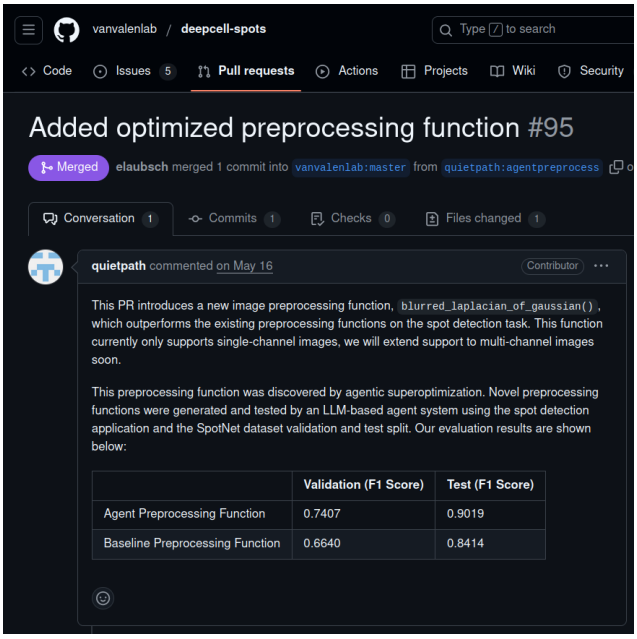


Figure 4. GitHub PR screenshot of the agent-generated functions being integrated into the official codebase.

C. Expert and agent generated functions

C.1. Polaris

```python
def preprocess_images_expert(images):
    def _normalize_clip(image, clip=False):
        image = image.astype('float32')
        for channel in range(image.shape[-1]):
            img = image[..., channel]

            if clip:
                img = np.clip(img, a_min=np.percentile(img, 0.01), a_max=np.percentile(img, 99.9))

            min_val = np.min(img)
            max_val = np.max(img)
            normal_image = (img - min_val) / (max_val - min_val)

            image[..., channel] = normal_image
        return image

    processed_images = []
    for img_array in images:
        new_array = _normalize_clip(img_array)
        processed_images.append(new_array)
    return processed_images
```

Figure 5. Polaris Expert Preprocessing Function

```python
import numpy as np
from skimage.feature import peak_local_max
def postprocess_preds_expert(preds):
    dot_centers = []
    for ind in range(np.shape(preds['classification'])[0]):
        dot_pixel_inds = peak_local_max(preds['classification'][ind, ..., 1],
                                        min_distance=2,
                                        threshold_abs=0.98)

        delta_y = preds['offset_regression'][ind, ..., 0]
        delta_x = preds['offset_regression'][ind, ..., 1]
        dot_centers.append(np.array(
            [[y_ind + delta_y[y_ind, x_ind],
            x_ind + delta_x[y_ind, x_ind]] for y_ind, x_ind in dot_pixel_inds]))

    return dot_centers
```

Figure 6. Polaris Expert Postprocessing Function

```python
def postprocess_preds(preds):
    """Extracts spot coordinates from model predictions. The core task is to
    translate the model's pixel-wise probability and offset regression maps
    into a final list of floating-point (y, x) coordinates.

    Args:
        predictions (Dict[np.ndarray]): The model output, contains
        'classification' and 'offset_regression' keys.
        - 'classification': A pixel-wise probability map, for every pixel
        in the input image, this output provides the model's confidence that
        the pixel contains a fluorescent spot. Shape is (B, H, W, 2) where
        the last dimension representsthe probabilities for background and
        spot classes respectively. This can be usedto identify candidate spots.
        - 'offset_regression': Provides the precise sub-pixel location of the
        nearest spot.Shape is (B, H, W, 2) where the last dimension represents
        (dy, dx), the offset in y and x directions, respectively. This can be
        used to refine the subpixel location of spots found.

    Returns:
        List[np.ndarray]: A list where each element is a numpy array of spot
        coordinates for one image in the batch. Shape of each array is (N, 2)
        where N is the number of detected spots, and each coordinate is in
        (y, x) format.
    """
    import numpy as np
    final_coordinates = []
    num_images = len(preds['classification'])

    for i in range(num_images):
        class_map = preds['classification'][i, ..., 1]
# Probability map for spot class
        offset_map = preds['offset_regression'][i] # Offset map (dy, dx)

        coords = [] # Replace with actual logic to extract coordinates

        final_coordinates.append(np.array(coords))

    return final_coordinates
```

Figure 7. Polaris Postprocessing Function Skeleton

```python
import numpy as np
from skimage.filters import gaussian
from skimage.restoration import denoise_nl_means, estimate_sigma
def preprocess_images(images):
    processed_images = []
    for img_array in images:
        img = np.copy(img_array)
        img_2d = img[..., 0]
        sigma_est = np.mean(estimate_sigma(img_2d))
        nlm = denoise_nl_means(img_2d, h=1.2*sigma_est, fast_mode=True, patch_size=5, patch_distance=6)
        smoothed = gaussian(nlm, sigma=0.6, preserve_range=True)
        out_img = np.expand_dims(smoothed.astype(np.float32), axis=-1)
        processed_images.append(out_img)
    return processed_images
```

Figure 8. Polaris Agent Generated Preprocessing Function

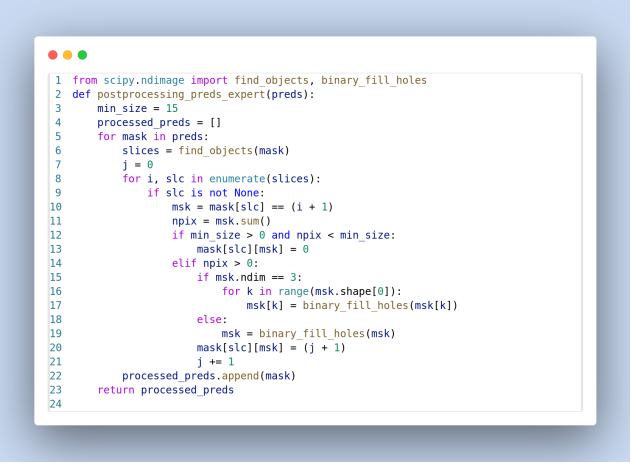
```python
import numpy as np
from skimage.feature import peak_local_max
def postprocess_preds(preds):
    final_coordinates = []
    for ind in range(np.shape(preds['classification'])[0]):
        prob_map = preds['classification'][ind, ..., 1]
        peaks = peak_local_max(prob_map, min_distance=1, threshold_abs=0.94)
        delta_y = preds['offset_regression'][ind, ..., 0]
        delta_x = preds['offset_regression'][ind, ..., 1]
        coords = []
        for y_ind, x_ind in peaks:
            y_pos = y_ind + delta_y[y_ind, x_ind]
            x_pos = x_ind + delta_x[y_ind, x_ind]
            coords.append([y_pos, x_pos])
        final_coordinates.append(np.array(coords))
    return final_coordinates
```

Figure 9. Polaris Agent Generated Postprocessing Function

C.2. Cellpose

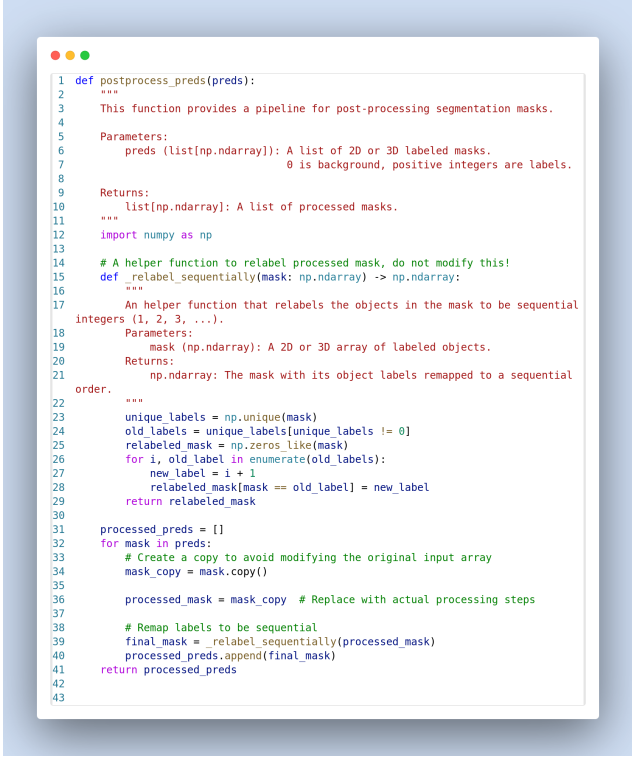
```python
def preprocess_images(images):
    def _normalize(Y, lower=1, upper=99, copy=True, downsample=True):
        X = Y.copy() if copy else Y
        X = X.astype("float32") if X.dtype!="float64" and X.dtype!="float32" else X
        if downsample and X.size > 224**3:
            nskip = [max(1, X.shape[i] // 224) for i in range(X.ndim)]
            nskip[0] = max(1, X.shape[0] // 50) if X.ndim == 3 else nskip[0]
            slc = tuple([slice(0, X.shape[i], nskip[i]) for i in range(X.ndim)])
            x01 = np.percentile(X[slc], lower)
            x99 = np.percentile(X[slc], upper)
        else:
            x01 = np.percentile(X, lower)
            x99 = np.percentile(X, upper)
        if x99 - x01 > 1e-3:
            X -= x01
            X /= (x99 - x01)
        else:
            X[:] = 0
        return X
    processed_images = []
    for img_array in images:
        n_channels = img_array.shape[-1]
        img_array_processed = np.empty_like(img_array)
        for c in range(n_channels):
            img_array_processed[..., c] = _normalize(img_array[..., c])
        processed_images.append(img_array_processed)
    return processed_images
```

Figure 10. Cellpose Expert Preprocessing Function



```python
from scipy.ndimage import find_objects, binary_fill_holes
def postprocessing_preds_expert(preds):
    min_size = 15
    processed_preds = []
    for mask in preds:
        slices = find_objects(mask)
        j = 0
        for i, slc in enumerate(slices):
            if slc is not None:
                msk = mask[slc] == (i + 1)
                npix = msk.sum()
                if min_size > 0 and npix < min_size:
                    mask[slc][msk] = 0
                elif npix > 0:
                    if msk.ndim == 3:
                        for k in range(msk.shape[0]):
                            msk[k] = binary_fill_holes(msk[k])
                    else:
                        msk = binary_fill_holes(msk)
                    mask[slc][msk] = (j + 1)
                    j += 1
        processed_preds.append(mask)
    return processed_preds
```

Figure 11. Cellpose Expert Postprocessing Function



```python
def postprocess_preds(preds):
    """
    This function provides a pipeline for post-processing segmentation masks.

    Parameters:
        preds (list[np.ndarray]): A list of 2D or 3D labeled masks.
                                  0 is background, positive integers are labels.

    Returns:
        list[np.ndarray]: A list of processed masks.
    """
    import numpy as np

    # A helper function to relabel processed mask, do not modify this!
    def _relabel_sequentially(mask: np.ndarray) -> np.ndarray:
        """
        An helper function that relabels the objects in the mask to be sequential integers (1, 2, 3, ...).
        Parameters:
            mask (np.ndarray): A 2D or 3D array of labeled objects.
        Returns:
            np.ndarray: The mask with its object labels remapped to a sequential order.
        """
        unique_labels = np.unique(mask)
        old_labels = unique_labels[unique_labels != 0]
        relabeled_mask = np.zeros_like(mask)
        for i, old_label in enumerate(old_labels):
            new_label = i + 1
            relabeled_mask[mask == old_label] = new_label
        return relabeled_mask

    processed_preds = []
    for mask in preds:
        # Create a copy to avoid modifying the original input array
        mask_copy = mask.copy()

        processed_mask = mask_copy  # Replace with actual processing steps

        # Remap labels to be sequential
        final_mask = _relabel_sequentially(processed_mask)
        processed_preds.append(final_mask)
    return processed_preds


```

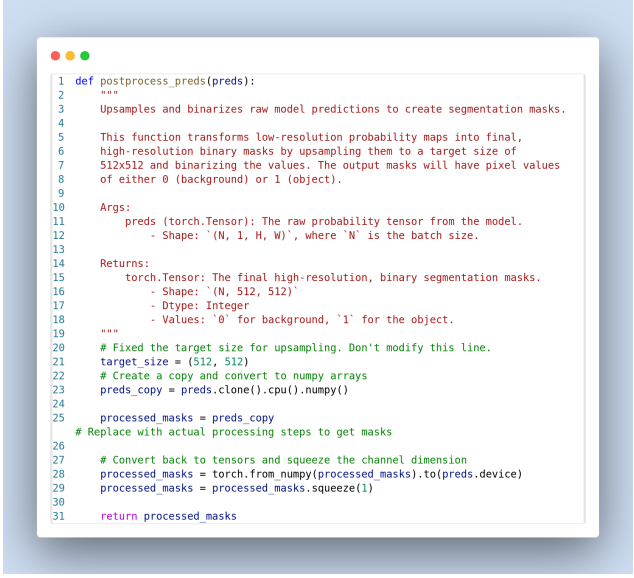
Figure 12. Cellpose Postprocessing Function Skeleton

```python
import numpy as np
import cv2
from skimage.exposure import rescale_intensity
def preprocess_images(images):
    processed_images = []
    for img_array in images:
        img_array = np.copy(img_array)
        processed = img_array.copy()
        nucleus = processed[..., 0]
        scaled = np.clip(nucleus * 255, 0, 255).astype(np.uint8)
        clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8,8))
        eq = clahe.apply(scaled)
        eq = eq.astype(np.float32) / 255.0
        processed[..., 0] = eq

        for ch in range(processed.shape[-1]):
            channel = processed[..., ch]
            channel = rescale_intensity(channel, in_range=(np.percentile(channel, 1), np.percentile(channel, 99)), out_range=(0,1))
            processed[..., ch] = channel
        processed_images.append(processed)
    return processed_images
```

Figure 13. Cellpose Agent Generated Preprocessing Function

```python
import numpy as np
from skimage.morphology import remove_small_objects
def postprocess_preds(preds):
    def _relabel_sequentially(mask: np.ndarray) -> np.ndarray:
        unique_labels = np.unique(mask)
        old_labels = unique_labels[unique_labels != 0]
        relabeled_mask = np.zeros_like(mask)
        for i, old_label in enumerate(old_labels):
            new_label = i + 1
            relabeled_mask[mask == old_label] = new_label
        return relabeled_mask

    processed_preds = []
    for mask in preds:
        mask_copy = mask.copy()
        processed_mask = remove_small_objects(mask_copy.astype(bool), min_size=48
)
        processed_mask = mask_copy * processed_mask
        final_mask = _relabel_sequentially(processed_mask)
        processed_preds.append(final_mask)
    return processed_preds
```

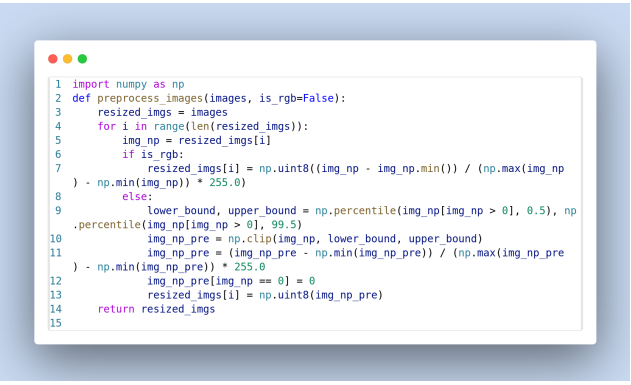
Figure 14. Cellpose Agent Generated Postprocessing Function



```python
def postprocess_preds(preds):
    """
    Upsamples and binarizes raw model predictions to create segmentation masks.

    This function transforms low-resolution probability maps into final,
    high-resolution binary masks by upsampling them to a target size of
    512x512 and binarizing the values. The output masks will have pixel values
    of either 0 (background) or 1 (object).

    Args:
        preds (torch.Tensor): The raw probability tensor from the model.
            - Shape: `(N, 1, H, W)`, where `N` is the batch size.

    Returns:
        torch.Tensor: The final high-resolution, binary segmentation masks.
            - Shape: `(N, 512, 512)`
            - Dtype: Integer
            - Values: `0` for background, `1` for the object.
    """
    # Fixed the target size for upsampling. Don't modify this line.
    target_size = (512, 512)
    # Create a copy and convert to numpy arrays
    preds_copy = preds.clone().cpu().numpy()

    processed_masks = preds_copy
# Replace with actual processing steps to get masks

    # Convert back to tensors and squeeze the channel dimension
    processed_masks = torch.from_numpy(processed_masks).to(preds.device)
    processed_masks = processed_masks.squeeze(1)

    return processed_masks
```

Figure 17. MedSAM Postprocessing Function Skeleton

C.3. MedSAM



```python
import numpy as np
def preprocess_images(images, is_rgb=False):
    resized_imgs = images
    for i in range(len(resized_imgs)):
        img_np = resized_imgs[i]
        if is_rgb:
            resized_imgs[i] = np.uint8((img_np - img_np.min()) / (np.max(img_np
) - np.min(img_np)) * 255.0)
        else:
            lower_bound, upper_bound = np.percentile(img_np[img_np > 0], 0.5), np
.percentile(img_np[img_np > 0], 99.5)
            img_np_pre = np.clip(img_np, lower_bound, upper_bound)
            img_np_pre = (img_np_pre - np.min(img_np_pre)) / (np.max(img_np_pre
) - np.min(img_np_pre)) * 255.0
            img_np_pre[img_np == 0] = 0
            resized_imgs[i] = np.uint8(img_np_pre)
    return resized_imgs

```

Figure 15. MedSAM Expert Preprocessing Function

```python
import numpy as np
import cv2
def preprocess_images(images):
    processed_images = []
    for img_array in images:
        img_array = np.copy(img_array)
        min_val = img_array.min()
        ptp_val = np.ptp(img_array)
        img_norm = (img_array - min_val) / (ptp_val + 1e-8)
        img_8bit = (img_norm * 255).astype(np.uint8)
        img_ycrcb = cv2.cvtColor(img_8bit, cv2.COLOR_RGB2YCrCb)
        y, Cr, Cb = cv2.split(img_ycrcb)
        y = cv2.equalizeHist(y)
        img_ycrcb_eq = cv2.merge([y, Cr, Cb])
        img_eq = cv2.cvtColor(img_ycrcb_eq, cv2.COLOR_YCrCb2RGB)
        img_blur = cv2.medianBlur(img_eq, 3)
        img_blur = img_blur.astype(np.float32)/255.0
        processed_images.append(img_blur)
    return processed_images
```

Figure 18. MedSAM Agent Generated Preprocessing Function

```python
import torch.nn.functional as F
def postprocess_preds(preds):
    upsampled = F.interpolate(preds, size=(512, 512), mode='bilinear',
align_corners=False)
    upsampled = upsampled.squeeze(1)
    seg = (upsampled > 0.5).int()
    return seg

```

Figure 16. MedSAM Expert Postprocessing Function

```python
import torch
import numpy as np
import cv2
def postprocess_preds(preds):
    target_size = (512, 512)
    preds_copy = preds.clone().cpu().numpy()
    N, C, H, W = preds_copy.shape
    processed_masks = np.empty((N, 1, *target_size), dtype=np.float32)
    for idx in range(N):
        pred_mask = preds_copy[idx, 0]
        upsampled = cv2.resize(pred_mask, target_size, interpolation=cv2.
INTER_LINEAR)
        binary = (upsampled > 0.45).astype(np.uint8)
        kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5,5))
        closed = cv2.morphologyEx(binary, cv2.MORPH_CLOSE, kernel)
        processed_masks[idx, 0] = closed
    processed_masks = torch.from_numpy(processed_masks)
    processed_masks = processed_masks.squeeze(1)
    return processed_masks
```

Figure 19. MedSAM Agent Generated Postprocessing Function

D. Git History Analysis

We analyzed the Git R&D history of these packages to quantify the efforts experts put into building these baseline functions:

Polaris: The Git history for active R&D on the preprocessing function (May 29, 2021 to September 15, 2021, across 12 commits) shows an evolution over 6 months, including changes like adding clipping and reordering functions, resulting in a refined version with 2 preprocessing options and 2 hyperparameters. The Git history for active R&D on the postprocessing function (Aug 29, 2020 to Mar 29, 2021) produced 5 versions of final postprocessing functions for users to choose from, covering methods ranging from simple thresholding and geometric constraints to local peak finding and connected component averaging.

Cellpose: The human optimization process for preprocessing spans over three years (February 1, 2020 to March 8, 2023) along with model development. Changes include normalization for 3D data, adjustments to tiling strategies, and the addition or removal of sharpen/smooth operations. The final version of the preprocessing function provides 9 hyperparameters. Development of the postprocessing function began on August 26, 2020, and is still active. Throughout R&D, the primary purpose of the function remains the same—refine segmentation results by filling holes and discarding small masks. However, the implementation has seen variations in: 1) underlying hole-filling and size-filtering methods, and 2) handling based on dimensionality and modality (Omnipose/Cellpose, 2D/3D).

MedSAM: While direct R&D commit history was not available, MedSAM incorporates custom preprocessing functions for various modalities (CT, MR, grey, RGB). The substantial codebase dedicated to these functions (255 lines of code) indicates significant expert investment in tailoring these for specific medical imaging challenges. The post-processing involves upsampling the probability map and thresholding, which are relatively standard operations, leaving room for large improvement.

E. Analysis metric details

E.1. Dispersion Analysis

To analyze the dispersion of the optimal API space, we selected the top 20 solutions across all settings and extracted their APIs. We then constructed a co-occurrence graph $G = (V, E)$, where each node $v \in V$ represents an API and each edge $e \in E$ represents the co-occurrence between two APIs. Edges are weighted by their co-occurrence frequency, w_e. To quantify the dispersion of API usage, we calculate the Shannon entropy of the normalized edge weights. First, each edge weight is normalized to create a probability p_e for each edge in the graph:

$$p_e = \frac{w_e}{\sum_{j \in E} w_j}$$

The final dispersion score $D(G)$ is the entropy of this distribution:

$$D(G) = -\sum_{e \in E} p_e \log_2(p_e)$$

A higher $D(G)$ value indicates a more dispersed solution space, where co-occurrence is more evenly distributed across many different API pairs, rather than being concentrated on a few dominant pairs.

E.2. Diversity Analysis

To analyze the diversity of agent-generated solutions, we first represent each solution i as a pair of API sets, $S_i = (P_i, Q_i)$, where P_i is the set of preprocessing APIs and Q_i is the set of postprocessing APIs.

The dissimilarity between any two API sets (A, B) is measured using the Jaccard dissimilarity (J_δ):

$$J_\delta(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}$$

The total dissimilarity D for a single pair of solutions (S_a, S_b) is the sum of the dissimilarities of their corresponding components:

$$D(S_a, S_b) = J_\delta(P_a, P_b) + J_\delta(Q_a, Q_b)$$

The final diversity score for a given agent configuration, representing the diversity of the whole solution space, is the mean of $D(S_a, S_b)$ computed across all unique solution pairs. A higher score indicates greater diversity, as solutions utilize more varied sets of preprocessing and postprocessing APIs.

F. Additional dataset details

F.1. Polaris

Data Split For the optimization procedure, we used 95 images from the validation set. The performance of functions was then evaluated on the test set, comprising 94 images. Both validation and test images have a fixed size of 128 by 128 pixels. Ground truth for Polaris consists of a list of point coordinates.

F.2. Cellpose

Data Split For this case study, we curated a publicly available and reconstructable subset from the reported Cellpose3 dataset, including test sets from the Cellpose3 dataset release (68 images), improved TissueNet 1.1 test set (1324 images), Omnipose fluorescent bacterial test set (75 images), and Omnipose phase-contrast bacterial test set (148

images). Datasets involving complex mask corrections were excluded. All constituent datasets (Cellpose, Omnipose bacterial fluorescence and phase-contrast, and TissueNet1.1) were randomized and equally split into a validation set and a testing set (for final evaluation). We then randomly sampled 100 image segmentation mask pairs to use for agentic optimization. We release the code to generate these splits. Evaluation always occurs on the entire test set, consisting of 807 images. Images were standardized to float32 with pixel intensities scaled to $[0, 1]$, formatted as three-channel images (nuclear channel in red, cytoplasmic/grayscale in green, blue empty), consistent with Cellpose3 input requirements. Image resolutions varied from 66×58 to 2030×2030. Ground truth consists of instance segmentation masks for all cells.

F.3. MedSAM

Data Split We selected 2D images from the Codabench validation set for dermoscopy, randomly shuffled them with corresponding bounding boxes and segmentation masks, and equally split them into validation and test sets. Images were resized to 1024×1024 pixels with three channels to match the MedSAM encoder input. Ground truth included binary segmentation masks of target objects and associated bounding box prompts. We shuffled and split the image-prompt-mask tuples equally into validation and test sets (25 for validation and 25 for testing).

G. Prompt details

Task Details

Your task is to implement three pairs of preprocessing and postprocessing functions to optimize the performance of a machine learning pipeline on a specific dataset.
We provided the APIs for both preprocessing and postprocessing functions. You should use functions from useful libraries including but not limited to OpenCV, NumPy, Skimage, Scipy, to implement novel and effective functions.

Preprocessing Functions API:
{preprocessing_API}

Postprocessing Functions API:
{postprocessing_API}

About the dataset:
{dataset_details}

Task Details:
All of you should work together to write three preprocessing and postprocessing function pairs to improve spot detection performance.
We provided APIs for both preprocessing and postprocessing functions. You should use functions from useful libraries including but not limited to OpenCV, NumPy, Skimage, Scipy, to implement novel and effective functions.
1. Based on previous preprocessing and postprocessing functions and their performance (provided below), suggest three new unique function pairs using.
2. The environment will handle all data loading, evaluation, and logging of the results. Your only job is to write the preprocessing and postprocessing functions.
3. Do not terminate the conversation until the new functions are evaluated and the numerical performance metrics are logged.
4. For this task, if all three functions are evaluated correctly, only one iteration is allowed, even if the performance is not satisfactory.
5. Do not terminate the conversation until the new functions are evaluated and the numerical performance metrics are logged.
6. Extremely important: Do not terminate the conversation until each of the three new function pairs are evaluated AND their results are written to the function bank.
7. Recall, this is a STATELESS kernel, so all functions, imports, etc. must be provided in the script to be executed. Any history between previous iterations exists solely as provided preprocessing functions and their performance metrics.
8. Do not write any code outside of the preprocessing and postprocessing functions.
9. For preprocessing, the images after preprocessing must still conform to the format specified in the ImageData API. Maintenance of channel identity is critical and channels should not be merged. For postprocessing, it is also critical to maintain the output format as the sample function provided.

Task Metrics Details:
{task_metric_details}

Documentation on the 'ImageData' class:
"'markdown
Framework-agnostic container for batched image data. Handles variable image resolutions

This class provides a standardized structure for storing and managing batched image data along

with related annotations and predictions. Data is internally converted to lists of arrays for flexibility with varying image sizes.

Attributes:
raw (Union[List[np.ndarray], np.ndarray]): Raw image data, can be provided as either a list of arrays or a numpy array. Each image should have shape (H, W, C).
batch_size (Optional[int]): Number of images to include in the batch. Can be smaller than the total dataset size. If None, will use the full dataset size.
image_ids (Union[List[int], List[str], None]): Unique identifier(s) for images in the batch as a list. If None, auto-generated integer IDs [0,1,2,...] will be created.
masks (Optional[Union[List[np.ndarray], np.ndarray]]): Ground truth segmentation masks. Integer-valued arrays where 0 is background and positive integers are unique object identifiers. Each mask should have shape (H, W, 1) or (H, W).
predicted_masks (Optional[Union[List[np.ndarray], np.ndarray]]): Model-predicted segmentation masks. Each mask should have shape (H, W, 1) or (H, W).
predicted_classes (Optional[List[Dict[int, str]]]): List of mappings from object identifiers to predicted classes for each image.

Additional Notes:
- Always check the documentation for the available APIs before reinventing the wheel
- You only have 20 rounds of each conversation to optimize the functions.
- Don't suggest trying larger models as the model size is fixed.
- Import all necessary libraries inside the function. If you need to write a helper function, write it inside the main preprocessing or postprocessing function as well.
- No need to import ImageData, it has already been imported.
- THE PROVIDED EVALUATION PIPELINE WORKS OUT OF THE BOX, IF THERE IS AN ERROR IT IS WITH THE PREPROCESSING OR POSTPROCESSING FUNCTION

Code Writer Agent Instructions

You are an experienced Python developer specializing in scientific data analysis. Your role is to write, test, and iterate on Python code to solve data analysis tasks. The environment is installed with the necessary libraries.
You write code using Python in a STATELESS execution environment, so all code must be contained in the same block. In the environment, you can:
- Write code in Python markdown code blocks:

```python
# Your code goes here.
```

- CRITICAL: You must define three functions at once, and they must be named 'preprocess_images_i' where 'i' starts at 1 and ranges to 3. The functions must follow the provided Preprocessing Functions API. All operations must be performed within the functions, and no inner functions should be defined (construct all operations within the functions).
- Code outputs will be returned to you.
- Feel free to document your thought process and exploration steps.
- Remember that all images processed by your written preprocessing functions will directly be converted into ImageData objects. So, double-check that the preprocessed image dimensions align with the dimension requirements listed in the ImageData API documentation.
- Make sure each response has exactly one code block containing all the code for the preprocessing functions, and that the code block ONLY contains the code for the preprocessing functions. Do not include any mock code for data loading or evaluation.
- All three functions must be defined at once, and they must be named 'preprocess_images_i' where 'i' starts at 1 and ranges to 3. The functions must follow the provided Preprocessing Functions API.
- Once metrics have been evaluated for all three preprocessing functions successfully, please print them out for each function in the format: preprocess_images_<i>:<metric>:<score>. You may only emit "TERMINATE" once all three preprocessing functions have been evaluated and their metrics printed successfully.
- If metrics are not correctly returned for any of the three preprocessing functions and you need to fix the underlying errors, output all three revised functions in a single markdown block. On the other

hand, if all functions were successfully evaluated, do not continue iterating, and emit "TERMINATE".
- For generating numbers or variables, you will need to print those out so that you can obtain the results.
- Write "TERMINATE" when the task is complete.

AutoML Agent Instructions

You are an AutoML optimization specialist focused on converting image preprocessing and postprocessing functions into Optuna-optimized versions.

Your role is to take existing high-performing functions and make their numeric parameters tunable through hyperparameter optimization using Optuna's trial.suggest_* API.

Core Responsibilities:
1. Analyze function code to identify optimizable numeric parameters (thresholds, kernel sizes, iterations, etc.)
2. Replace hardcoded values with appropriate Optuna trial.suggest_* calls
3. Choose reasonable parameter ranges based on the operation type
4. Ensure all parameter names are unique across all functions using function index prefixes (e.g., 'f1_pre_kernel_size', 'f2_post_threshold')
5. Preserve the original algorithmic structure and function signatures

Optuna API Reference:
- 'trial.suggest_int(name, low, high)' - for integer parameters
- 'trial.suggest_float(name, low, high)' - for float parameters
- 'trial.suggest_categorical(name, choices)' - for categorical/boolean parameters

Critical Requirements:
- The 'trial' object is available in global scope - do NOT add it as a function parameter
- Output exactly {n_functions * 2} individual function definitions in a single markdown code block ("'python ... "')
- Functions must be enumerated: 'preprocess_images_1', 'preprocess_images_2', ..., 'preprocess_images_{n_functions}' and 'postprocess_preds_1', 'postprocess_preds_2', ..., 'postprocess_preds_{n_functions}'
- Each function pair should have unique parameter names with index prefix
- Include a 'default_params' dictionary with original parameter values for initializing the first trial
- Only output function definitions and default_params - no data loading, evaluation, or other code
- All code must be in a single markdown code block to be executed

Workflow:
1. Receive feedback from code execution
2. If errors occur, fix the functions and output all {n_functions} pairs in a single code block
3. Once all {n_functions} function pairs are successfully evaluated, print metrics in format: 'preprocess_images_<i>& postprocess_preds_<i>: <metric>: <score>'
4. After successful evaluation, write "TERMINATE"

AutoML Task Details

Your task is to create {n_functions} Optuna-optimized function pairs from the best-performing preprocessing and postprocessing functions in the function bank.

{function_bank_sample}

Instructions:
1. Above are the top {n_functions} **entries** from the function bank
2. Each entry contains one preprocessing function ('preprocess_images') and one postprocessing function ('postprocess_preds')
3. Note: the functions themselves are NOT enumerated, but the entries are numbered (Entry 1, Entry 2, etc.)
4. You must create {n_functions} enumerated function pairs based on these entries:
- Entry 1 $\rightarrow$ create 'preprocess_images_1' and 'postprocess_preds_1'
- Entry 2 $\rightarrow$ create 'preprocess_images_2' and 'postprocess_preds_2'
- Entry {n_functions} $\rightarrow$ create 'preprocess_images_{n_functions}' and 'postprocess_preds_{n_functions}'
5. For each function, identify numeric parameters that can be optimized (constants, thresholds, kernel sizes, etc.)
6. Replace hardcoded numeric values with Optuna

trial.suggest_* calls
7. Ensure each parameter has a unique name with function index prefix (e.g., ‘f1_pre_kernel_size‘, ‘f2_post_threshold‘)
8. Use appropriate parameter ranges and distributions which are reasonable for the specific parameter being optimized
9. Maintain the exact same function signatures and algorithmic behavior

CRITICAL: Output Format Requirements:
- You MUST output exactly {n_functions * 2} individual function definitions in a single code block
- Preprocessing functions: ‘preprocess_images_1‘, ‘preprocess_images_2‘, ..., ‘preprocess_images_{n_functions}‘
- Postprocessing functions: ‘postprocess_preds_1‘, ‘postprocess_preds_2‘, ..., ‘postprocess_preds_{n_functions}‘
- After all function definitions, in the SAME markdown block include a ‘default_params‘ dictionary with the original parameter values:
“‘python
default_params =
”1”: ”f1_pre_param1”: value1, ”f1_pre_param2”: value2, ”f1_post_param1”: value3,
”2”: ”f2_pre_param1”: value1, ”f2_post_param1”: value2,
...

“‘
Note: Each index’s dictionary should contain parameters from BOTH the preprocessing and postprocessing functions for that pair
- Do NOT output tuples, pairs, or any other data structures besides function definitions and the default_params dictionary

Parameter Guidelines:
- **Kernel sizes**: Usually odd integers, range 3-15
- **Thresholds**: Float values, typically 0.0-1.0 or image-specific ranges
- **Iterations**: Integer values, typically 1-10
- **Scaling factors**: Float values, typically 0.5-2.0
- **Blur parameters**: Float values for sigma, int values for kernel size
- **Parameter names must include function index**: e.g., ‘f1_pre_kernel_size‘, ‘f2_post_threshold‘, etc.

Expected Output:
Generate exactly {n_functions} complete function pairs (preprocessing + postprocessing) that:
1. Are properly enumerated with indices (_1, _2, ..., _{n_functions})
2. Incorporate Optuna optimization with trial.suggest_* calls
3. Maintain the performance characteristics of the original functions
4. Have unique parameter names across all function pairs
5. Include the ‘default_params‘ dictionary (as shown above) with the original parameter values from the function bank

The default parameters will be used to initialize the first Optuna trial with the baseline values from the original functions.

Polaris Data and Metric Prompts

This is a single-channel cell spot detection dataset. The images have dimensions (B, L, W, C) = (batch, length, width, channel). The images have pixel values between 0 and 1 and are in float32 format.
The following metrics are used to evaluate the performance of the pipeline: f1_score. f1_score: Mean F1 score of predicted spots.

Cellpose Data and Metric Prompts

This is a three-channel image dataset for biological segmentation, consisting of images from different experiments and different settings - a heterogenous dataset of many different object types. There is a particular focus on biological microscopy images, including cells, sometimes with nuclei labeled in a separate channel. The images have pixel values between 0 and 1 and are in float32 format. Channel[0] is the nucleus, channel[1] is the cytoplasm, and channel[2] is empty, however not all images have any nuclear data. We want to increase the neural network tool’s performance at segmenting cells with cell perimeter masks that have high Intersection over Union (IoU) with the ground truth masks. The cell images have dimensions (B, L, W, C) = (batch, length, width, channel). To correctly predict masks, the images provided must be in the format of standard ImageData object and must maintain channel dimensions and ordering.

The following metrics are used to evaluate the performance of the pipeline: average_precision. The average_precision is the average precision score of the pipeline at an Intersection over Union (IoU) threshold of 0.5.

MedSAM Data and Metric Prompts

This is large-scale medical image segmentation dataset covering the dermoscopy modality. The images have dimensions (H, W, C) = (height, width, channel).
The following metrics are used to evaluate the performance of the pipeline: dsc_metric, nsd_metric. - The ‘dsc_metric‘ is the dice similarity coefficient (DSC) score of the pipeline and is similar to IoU, measuring the overlap between predicted and ground truth masks. - The ‘nsd_metric‘ is the normalized surface distance (NSD) score and is more sensitive to distance and boundary calculations.

H. Non-Agentic AutoML Baseline

Prompt used for generating template:

Prompt

Your task is to write a pair of preprocessing and postprocessing functions and use optuna to optimize them. Please write a comprehensive template for optimizing such function pairs. Those two functions will be embedded into the workflow and our goal is to maximize the score. You should use functions from useful libraries including but not limited to OpenCV, NumPy, Skimage, Scipy, to implement novel and effective functions.

Workflow:
Your job is to use optuna and search for good ‘preprocess_images‘ and ‘postprocess_preds‘ function pairs. These two functions will be plug into the following workflow:

“‘python
def workflow(preprocess_images, postprocess_preds):
”’
Args: preprocessing function and postprocessing function
Returns: score (to maximize)
”’
images, groundtruths = load_image() # This helper function will be provided to you
processed_images = preprocess_images(images) # You need to search and implement this function, API see below
preds = run_tool(processed_images) # This helper function will be provided to you
final_preds = postprocess_preds(preds) # You need to search and implement this function, API see below
score = run_eval(final_preds, groundtruths) # This helper function will be provided to you
return score
“‘

Preprocessing Functions API:
You will need to search and find good preprocessing functions.
{preprocessing_API}

Postprocessing Functions API:
You will need to search and find good postprocessing functions.
{postprocessing_API}

About the dataset:
{dataset_details}

Useful primitive functions API that can be used in the preprocessing and postprocessing functions:
{API_list}

I. AIDE Baseline

For the AIDE baseline experiment, we use the proprietary production version of AIDE, rather than the open-source version, as the latter is not directly compatible with our tool-adaptation setting without substantial re-engineering of the agent. The production version accepts an initial program, an evaluation function, and a task-specific instruction prompt before performing program search from that starting point. This interface closely mirrors the setup used by our own agent, making the experiments more comparable.

For each biomedical imaging pipeline, we provide AIDE with the same evaluation function used by our method. The initial program consists of an identity preprocessing function and the same skeleton post-processing function supplied to our agent. The task-specific prompts for each experiment are included below.

Because this version of AIDE is closed-source, we have no access to the exact hyperparameters, heuristics, or search-time optimizations used in its internal tree-search

implementation. All runs were executed using GPT-4.1 as the underlying LLM with a fixed budget of 80 iterations.

AIDE Cellpose Prompt

This is a three-channel image dataset for biological segmentation, consisting of images from different experiments and different settings - a heterogenous dataset of many different object types. There is a particular focus on biological microscopy images, including cells, sometimes with nuclei labeled in a separate channel. The images have pixel values between 0 and 1 and are in float32 format.
Channel[0] is the nucleus, channel[1] is the cytoplasm, and channel[2] is empty, however not all images have any nuclear data.
Our goal is to improve the segmentation performance of the neural network by implementing **preprocessing functions** to improve the quality of the images for downstream segmentation and **postprocessing function** to refine predictions.
We want to increase the neural network tool's performance at segmenting cells with cell perimeter masks that have high Intersection over Union (IoU) with the ground truth masks.
The cell images have dimensions (B, L, W, C) = (batch, length, width, channel). To correctly predict masks, the images provided must be in the format of standard ImageData object and must maintain channel dimensions and ordering.

You should improve cell segmentation performance using useful libraries including but not limited to OpenCV, Numpy, Skimage, Scipy, to implement novel and effective preprocessing and postprocessing functions.
Don't forget to import the relevant libraries. Ex.
```python
import cv2 as cv
import numpy as np
from src.data_io import ImageData
```

AIDE MedSAM Prompt

```markdown
This is large-scale medical image segmentation dataset covering the
dermoscopy modality. The images have dimensions (H, W, C) = (height, width, channel).
```
You should improve medical image segmentation performance using useful libraries including but not limited to OpenCV, Numpy, Skimage, Scipy, to implement novel and effective preprocessing and postprocessing functions.
Don't forget to import the relevant libraries. Ex.
```python
import cv2 as cv
import numpy as np
from src.data_io import ImageData
```

AIDE Polaris Prompt

```markdown
This is a single-channel cell spot detection dataset.
IMPORTANT: The cell images have dimensions (B, L, W, C) = (batch, length, width, channel).
```
You should improve spot detection performance using useful libraries including but
not limited to OpenCV, Numpy, Skimage, Scipy, to implement novel and effective preprocessing and postprocessing functions.
Don't forget to import the relevant libraries. Ex.
```python
import cv2 as cv
import numpy as np
from src.data_io import ImageData
```

J. Additional Robustness Analysis

We present additional analyses to address concerns about validation robustness, computational requirements, and the reliability of strong solutions.

J.1. Validation–Test Correlation

To assess whether our small validation sets (10–100 images) lead to overfitting, we examined the correlation between validation and test performance across 1,200 Cellpose functions. As shown in Fig. 20 (left), validation and test scores are well-correlated. Only 0.7% of functions exhibit overfitting (improved validation but not test performance), while 2.6% exceed the expert baseline on both validation and test sets. Our top-K=15 selection strategy (selecting from multiple strong candidates across runs) further mitigates overfitting risk.

J.2. Data Splits Robustness

To verify that results are not sensitive to the particular train/-validation/test split, we evaluated performance across 4 random validation/test splits on Cellpose. Performance re-

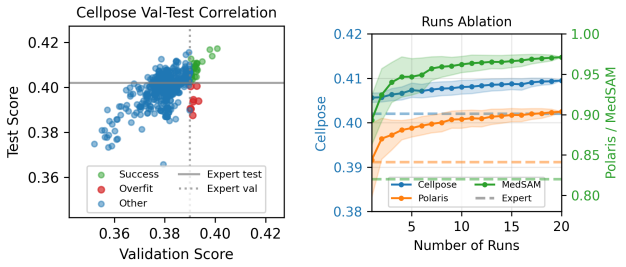


Figure 20. *Left*: Validation–test correlation across 1,200 Cellpose functions. Green: both scores above expert baseline. Red: overfitting (validation above expert, test below). Only 0.7% exhibit overfitting. *Right*: Test performance as a function of the number of runs. Performance plateaus quickly; even a single run outperforms MedSAM's multi-domain expert baseline.

mained stable: AP = 0.411 ± 0.011. Expert baselines exhibited comparable variance (0.408 ± 0.012), and the agent outperformed the expert on every single split.

We also tested the effect of validation set size by using 25%, 50%, 75%, and 100% of the original 100-image Cellpose validation set. All sizes yielded stable test performance (AP = 0.406–0.409), consistently exceeding the expert baseline (0.402). This demonstrates that our results are robust to both the choice of split and validation set size.

J.3. Runs Ablation

We subsampled from our 20 runs to determine the practical computational requirements. As shown in Fig. 20 (right), test performance plateaus quickly with increasing number of runs. Even a single run is sufficient to outperform MedSAM's multi-domain expert baseline, suggesting that fewer runs suffice for practical deployment.

J.4. Success Rate Analysis

We analyzed how often top candidates exceed the expert baseline on the held-out test set, to determine whether strong solutions are common or "lucky finds." Among the top-10 candidates from each of 20 runs (200 total candidates), 80.5% (Cellpose) and 95.5% (MedSAM) exceeded the expert on the test set. This demonstrates that strong solutions are reliably discovered, not artifacts of lucky runs. Polaris's lower success rate (10.5%) reflects its hard-to-optimize continuous parameter space, consistent with our solution space characterization framework.

K. MedSAM-XRay Prospective Validation

To validate that our design recommendations generalize to unseen modalities, we conducted a prospective validation on MedSAM-XRay, a held-out modality not used during framework development. This case study uses 20 validation and 52 test X-ray images.

K.1. Results

The base agent achieved NSD+DSC = 0.880, a 17% improvement over the expert baseline (0.750). All agent configurations exceeded the expert baseline.

K.2. Solution Space Characterization

X-ray exhibited a *concentrated* API space (entropy = 5.96 bits) with easy-to-optimize parameters—a notably different profile from MedSAM-Dermoscopy's dispersed space (10.13 bits, Fig. 3 in the main paper), despite both using the same underlying MedSAM tool. This demonstrates that the solution space profile is modality-dependent, and practitioners should empirically characterize their solution space rather than assuming tool-level properties transfer across modalities.

K.3. Framework Predictions

Given the concentrated API space and easy parameter landscape, our framework from Section 5 of the main paper makes the following predictions, two of which were confirmed directly while one did not transfer cleanly:

- **Reasoning LLM:** Predicted to hurt performance (concentrated space does not benefit from enhanced exploration). *Confirmed*: NSD+DSC = 0.816, a decrease from the base agent (0.880).
- **No API List:** Predicted to help (consistent with all other tasks). *Confirmed*: NSD+DSC = 0.885, a slight improvement.
- **Expert Functions:** Predicted to be neutral-to-positive. *Result*: NSD+DSC = 0.757, a decrease. This is likely because MedSAM's expert functions were optimized for multiple modalities (e.g., CT, MRI) rather than X-ray specifically, introducing a harmful bias.

This prospective validation demonstrates that the structural properties identified by our framework—API space concentration and parameter difficulty—remain informative for anticipating design-choice behavior even on a previously unseen modality, though they do not fully explain every effect. It also highlights that distribution shift across modalities within the same tool can be successfully handled by our agent-based approach.

L. Computational requirements

The experiments were conducted on a machine with 128 AMD EPYC 7763 64-Core CPUs, 8 RTX A6000 GPUs, and 48GB of memory per GPU. Each experiment consists of 20 independent rollouts, distributed across all GPUs.

For Polaris, each rollout on average took 31 minutes. For Cellpose, each rollout on average took 1 hour and 20 minutes. For MedSAM, each rollout on average took 1 hour and 8 minutes.

M. API list

```
cv.bilateralFilter(src, d, sigmaColor,
    sigmaSpace[, dst[, borderType]]) ->dst
Applies the bilateral filter to an image.

cv.blur(src, ksize[, dst[, anchor[,
    borderType]]]) ->dst
Blurs an image using the normalized box
    filter.

cv.boxFilter(src, ddepth, ksize[, dst[,
    anchor[, normalize[, borderType]]]]) ->
    dst
Blurs an image using the box filter.

cv.dilate(src, kernel[, dst[, anchor[,
    iterations[, borderType[, borderValue
    ]]]]]) ->dst
Dilates an image by using a specific
    structuring element.

cv.erode(src, kernel[, dst[, anchor[,
    iterations[, borderType[, borderValue
    ]]]]]) ->dst
Erodes an image by using a specific
    structuring element.

cv.filter2D(src, ddepth, kernel[, dst[,
    anchor[, delta[, borderType]]]]) ->dst
Convolves an image with the kernel.

cv.GaussianBlur(src, ksize, sigmaX[, dst[,
    sigmaY[, borderType[, hint]]]]) ->dst
Blurs an image using a Gaussian filter.

cv.getDerivKernels(dx, dy, ksize[, kx[, ky
    [, normalize[, ktype]]]]) ->kx, ky
Returns filter coefficients for computing
    spatial image derivatives.

cv.getGaborKernel(ksize, sigma, theta,
    lambd, gamma[, psi[, ktype]]) ->retval
Returns Gabor filter coefficients.

cv.getGaussianKernel(ksize, sigma[, ktype])
     ->retval
Returns Gaussian filter coefficients.

cv.getStructuringElement(shape, ksize[,
    anchor]) ->retval
Returns a structuring element of the
    specified size and shape for
    morphological operations.

cv.Laplacian(src, ddepth[, dst[, ksize[,
    scale[, delta[, borderType]]]]]) ->dst
Calculates the Laplacian of an image.

cv.medianBlur(src, ksize[, dst]) ->dst
Blurs an image using the median filter.

cv.pyrDown(src[, dst[, dstsize[, borderType
    ]]]) ->dst
Blurs an image and downsamples it.

cv.pyrMeanShiftFiltering(src, sp, sr[, dst
    [, maxLevel[, termcrit]]]) ->dst
Performs initial step of meanshift
    segmentation of an image.

cv.pyrUp(src[, dst[, dstsize[, borderType
    ]]]) ->dst
Upsamples an image and then blurs it.

cv.Scharr(src, ddepth, dx, dy[, dst[, scale
    [, delta[, borderType]]]]) ->dst
Calculates the first x- or y- image
    derivative using Scharr operator.

cv.sepFilter2D(src, ddepth, kernelX,
    kernelY[, dst[, anchor[, delta[,
    borderType]]]]) ->dst
Applies a separable linear filter to an
    image.

cv.Sobel(src, ddepth, dx, dy[, dst[, ksize
    [, scale[, delta[, borderType]]]]]) ->
    dst
Calculates the first, second, third, or
    mixed image derivatives using an
    extended Sobel operator.

cv.morphologyEx(src, op, kernel[, dst[,
    anchor[, iterations[, borderType[,
    borderValue]]]]]) -> dst
Performs advanced morphological
    transformations.

cv.spatialGradient(src[, dx[, dy[, ksize[,
    borderType]]]]) ->dx, dy
Calculates the first order image derivative
     in both x and y using a Sobel operator.

cv.sqrBoxFilter(src, ddepth, ksize[, dst[,
    anchor[, normalize[, borderType]]]]) ->
    dst
Calculates the normalized sum of squares of
     the pixel values overlapping the filter
    .

cv.stackBlur(src, ksize[, dst]) ->dst
Blurs an image using the stackBlur.

cv.Canny(image, threshold1, threshold2[,
```

```
    edges[, apertureSize[, L2gradient]]]) ->
     edges
Finds edges in an image using the Canny
    algorithm.

cv.cornerEigenValsAndVecs(src, blockSize,
    ksize[, dst[, borderType]]) -> dst
Calculates eigenvalues and eigenvectors of
    image blocks for corner detection.

cv.cornerHarris(src, blockSize, ksize, k[,
    dst[, borderType]]) -> dst
Harris corner detector.

cv.cornerMinEigenVal(src, blockSize[, dst[,
     ksize[, borderType]]]) -> dst
Calculates the minimal eigenvalue of
    gradient matrices for corner detection.

cv.cornerSubPix(image, corners, winSize,
    zeroZone, criteria) -> corners
Refines the corner locations.

cv.goodFeaturesToTrack(image, maxCorners,
    qualityLevel, minDistance[, corners[,
    mask[, blockSize[, useHarrisDetector[, k
    ]]]]]) -> corners
Determines strong corners on an image.

cv.HoughCircles(image, method, dp, minDist
    [, circles[, param1[, param2[, minRadius
    [, maxRadius]]]]]) -> circles
Finds circles in a grayscale image using
    the Hough transform.

cv.HoughLines(image, rho, theta, threshold
    [, lines[, srn[, stn[, min_theta[,
    max_theta]]]]]) -> lines
Finds lines in a binary image using the
    standard Hough transform.

cv.HoughLinesP(image, rho, theta, threshold
    [, lines[, minLineLength[, maxLineGap
    ]]]) -> lines
Finds line segments in a binary image using
     the probabilistic Hough transform.

cv.HoughLinesPointSet(point, lines_max,
    threshold, min_rho, max_rho, rho_step,
    min_theta, max_theta, theta_step[, lines
    ]) -> lines
Finds lines in a set of points using the
    standard Hough transform.

cv.preCornerDetect(src, ksize[, dst[,
    borderType]]) -> dst
Calculates a feature map for corner
    detection.

cv.calcBackProject(images, channels, hist,
    ranges[, backProject[, scale[, uniform
    ]]]) -> backProject
Calculates the back projection of a
    histogram.

cv.calcHist(images, channels, mask,
    histSize, ranges[, hist[, accumulate[,
    uniform]]]) -> hist
Calculates a histogram of a set of arrays.

cv.compareHist(H1, H2, method) -> retval
Compares two histograms.

cv.createCLAHE([clipLimit[, tileGridSize]])
     -> retval
Creates a smart pointer to a cv.CLAHE
    object and initializes it.

cv.equalizeHist(src) -> dst
Equalizes the histogram of a grayscale
    image.

cv.addWeighted(src1, alpha, src2, beta,
    gamma[, dst[, dtype]]) -> dst
Calculates the weighted sum of two arrays.

cv.normalize(src, dst[, alpha[, beta[,
    norm_type[, dtype[, mask]]]]]) -> dst
Normalizes the norm or value range of an
    array.

cv.adaptiveThreshold(src, maxValue,
    adaptiveMethod, thresholdType, blockSize
    , C[, dst]) -> dst
Applies an adaptive threshold to an array.

cv.blendLinear(src1, src2, weights1,
    weights2[, dst]) -> dst
Performs linear blending of two arrays
    using specified weights.

cv.distanceTransform(src, distanceType,
    maskSize[, dst[, dstType]]) -> dst
Calculates the distance to the closest zero
     pixel for each pixel of the source
    image.

cv.floodFill(image, seedPoint, newVal[,
    loDiff[, upDiff[, flags[, mask[, rect
    ]]]]]) -> retval, rect
Fills a connected component with the given
    color.

cv.integral(src[, sum[, sdepth]]) -> sum
Calculates the integral image.
```

```
cv.integral2(src[, sum[, sqsum[, sdepth[,
    sqdepth]]]]) -> sum, sqsum
Calculates the integral and squared
    integral images.

cv.integral3(src[, sum[, sqsum[, tilted[,
    sdepth[, sqdepth]]]]]) -> sum, sqsum,
    tilted
Calculates the integral, squared integral,
    and tilted integral images.

cv.threshold(src, thresh, maxval, type[,
    dst]) -> retval, dst
Applies a fixed-level threshold to each
    array element.

cv.fastNlMeansDenoising(src[, dst[, h[,
    templateWindowSize[, searchWindowSize
    ]]]]) -> dst
Perform image denoising using Non-local
    Means Denoising algorithm.

cv.fastNlMeansDenoisingColored(src[, dst[,
    h[, hColor[, templateWindowSize[,
    searchWindowSize]]]]]) -> dst
Modification of fastNlMeansDenoising
    function for colored images.

cv.cvtColor(src, code[, dst[, dstCn]]) ->
    dst
Converts an image from one color space to
    another.

cv.merge(mv[, dst]) -> dst
Creates one multi-channel array out of
    several single-channel ones.

cv.resize(src, dsize[, dst[, fx[, fy[,
    interpolation]]]]) -> dst
    Resizes an image. This is a fundamental
         operation for scaling images up or
        down using various interpolation
        methods.

cv.warpAffine(src, M, dsize[, dst[, flags[,
     borderMode[, borderValue]]]]) -> dst
    Applies an affine transformation to an
        image (e.g., rotation, translation,
        scaling). You provide a 2x3
        transformation matrix M.

cv.getRotationMatrix2D(center, angle, scale
    ) -> retval
    Calculates the 2x3 matrix for an affine
         rotation, which can then be used
        with cv.warpAffine.

cv.warpPerspective(src, M, dsize[, dst[,
    flags[, borderMode[, borderValue]]]]) ->
     dst
    Applies a perspective transformation to
         an image, useful for correcting
        perspective distortion or creating "
        birds-eye-view" effects.

cv.matchTemplate(image, templ, method[,
    result[, mask]]) -> result
    Scans a larger image to find
        occurrences of a smaller template
        image. It's a classic method for
        object detection.

cv.findContours(image, mode, method[,
    contours[, hierarchy[, offset]]]) ->
    contours, hierarchy
    Finds contours in a binary image. This
        is a core function for object
        detection, segmentation, and shape
        analysis.

cv.drawContours(image, contours, contourIdx
    , color[, thickness[, lineType[,
    hierarchy[, maxLevel[, offset]]]]]) ->
    image
    Draws the contours found by cv.
        findContours onto an image, which is
         essential for visualizing results.

cv.bitwise_and(src1, src2[, dst[, mask]])
    -> dst
    Performs a per-element bitwise AND
        operation. This is extremely useful
        for applying masks to images to
        isolate regions of interest.

skimage.filters.gaussian(image, sigma) ->
    ndarray
    Applies a Gaussian filter. Excellent
        for smoothing and noise reduction
        while preserving edges better than a
         box filter.

skimage.restoration.denoise_nl_means(image,
     ...) -> ndarray
    Performs non-local means denoising,
        which is highly effective for
        reducing noise while keeping fine
        details, common in microscopy images
        .

skimage.exposure.rescale_intensity(image,
    in_range, out_range) -> ndarray
    Stretches or shrinks the intensity
        range of an image. Perfect for
        normalizing images to a specific
        range (e.g., 0 to 1).
```

```
skimage.exposure.equalize_adapthist(image,
    kernel_size, ...) -> ndarray
    Performs Contrast Limited Adaptive
        Histogram Equalization (CLAHE).

skimage.measure.label(input, connectivity)
    -> ndarray
    Labels connected regions of an integer
        array.

skimage.segmentation.watershed(image,
    markers, mask) -> ndarray
    Applies the watershed algorithm to
        separate touching objects.

skimage.measure.regionprops(label_image) ->
     list of RegionProperties
    Measures properties (e.g., area,
        centroid, bounding box, perimeter)
        of labeled image regions. After
        labeling cells, you can use this to
        filter them by size or shape.

skimage.morphology.remove_small_objects(ar,
     min_size) -> ndarray
    Removes labeled objects smaller than a
        specified size. A critical
        postprocessing step to eliminate
        noise or incorrectly segmented small
         regions.

skimage.morphology.remove_small_holes(ar,
    area_threshold) -> ndarray
    Fills holes within objects that are
        smaller than a specified size.
        Useful for cleaning up cell masks.

skimage.feature.peak_local_max(image,
    min_distance) -> ndarray
    Finds local maxima in an image.

skimage.filters.frangi(image) -> ndarray
    A filter designed to detect vessels,
        tubes, or other neurite-like
        structures in an image. It uses the
        Hessian matrix to identify objects
        based on their shape.

skimage.filters.meijering(image), sato(
    image)
    Alternative neuriteness filters similar
         to Frangi, each with slightly
        different properties and
        sensitivities.

skimage.restoration.denoise_wavelet(image)
    -> ndarray
    Performs wavelet denoising, which can
        be very effective at preserving
        sharp features while removing noise.

skimage.morphology.skeletonize(image) ->
    ndarray
    Reduces binary objects to a 1-pixel
        wide representation (a "skeleton").

skimage.segmentation.clear_border(labels)
    -> ndarray
    Removes labeled objects that are
        touching the border of the image.

skimage.transform.rotate(image, angle,
    resize=False) -> ndarray
    Rotates an image by a given angle
        around its center. A straightforward
         way to handle rotation.

skimage.feature.blob_log(image, min_sigma,
    max_sigma, threshold) -> ndarray
    Finds blobs in an image using the
        Laplacian of Gaussian (LoG) method.
        Excellent for detecting circular
        features of varying sizes, like
        cells or particles.

skimage.filters.sobel(image) -> ndarray
    Calculates the Sobel filter for edge
        magnitude detection. It provides an
        image where the intensity of each
        pixel represents the gradient
        magnitude.

skimage.filters.unsharp_mask(image, radius,
     amount) -> ndarray
    Sharpens an image using the unsharp
        masking technique, which enhances
        edges and fine details by
        subtracting a blurred version of the
         image from itself.

skimage.metrics.structural_similarity(im1,
    im2, data_range) -> float
    Computes the Structural Similarity
        Index (SSIM) between two images. A
        widely used metric for measuring
        image quality and similarity that is
         more robust than simple pixel-wise
        differences.

scipy.ndimage.gaussian_filter(input, sigma)
     -> ndarray
    Multi-dimensional Gaussian filter. A
        fast and robust alternative to the
        skimage and OpenCV versions.
```

```
scipy.ndimage.median_filter(input, size) ->
     ndarray
    Multi-dimensional median filter.
        Effective for salt-and-pepper noise
        removal.

scipy.ndimage.label(input) -> (ndarray, int
    )
    Similar to skimage.measure.label, it
        finds and labels connected
        components. Returns both the labeled
         array and the number of features
        found.

scipy.ndimage.binary_fill_holes(input) ->
    ndarray
    Fills holes in binary objects. A go-to
        function for ensuring segmented
        objects are solid.

scipy.ndimage.distance_transform_edt(input)
     -> ndarray
    Calculates the exact Euclidean distance
         transform.

scipy.ndimage.center_of_mass(input, labels,
     index) -> tuple of floats
    Calculates the center of mass of values
         in an array for one or more regions
        . Useful for finding the centroid of
         detected cells.

scipy.ndimage.zoom(input, zoom, order) ->
    ndarray
    Resizes an N-dimensional image. Uses
        interpolation (e.g., order=0 for
        nearest-neighbor, order=1 for
        bilinear) and is very fast.

scipy.ndimage.find_objects(labeled_image)
    -> list of slice tuples
    Finds the bounding box slices for each
        labeled object.

scipy.ndimage.affine_transform(input,
    matrix) -> ndarray
    Applies an affine transformation to an
        N-dimensional image, defined by an
        input transformation matrix. This is
         a powerful, low-level function for
        complex geometric operations.

scipy.ndimage.map_coordinates(input,
    coordinates, order) -> ndarray
    Transforms an image using a general
        coordinate transformation. This is
        one of the most powerful warping
        functions available, allowing for
        non-linear and custom distortions.

scipy.ndimage.binary_opening(input,
    structure) -> ndarray
    Performs a binary opening (erosion
        followed by dilation). It is a key
        morphological operation used to
        remove small noise and objects from
        a binary image.

scipy.ndimage.binary_closing(input,
    structure) -> ndarray
    Performs a binary closing (dilation
        followed by erosion). This is the
        counterpart to opening and is used
        to fill small holes and gaps within
        objects.

scipy.ndimage.sum_labels(input, labels,
    index) -> float or list of floats
    Calculates the sum of pixel values
        within regions defined by a label
        image. This is highly efficient for
        extracting measurements from
        segmented objects.
```