Appendix

A. Additional Implementation Details

In this section, we provide additional details of the data preparation and pose-processing for different tasks. PyTorch-style pseudo-code is provided to better illustrate the implementation details. They can be done with a few operations in implementation. For each task, Painter doesn't involve more complex post-processing compared to the specialist methods.

Semantic segmentation As described in Section 3.1 of the main paper, we formulate different semantic categories using different colors in RGB space. To this end, we define the background and ignore areas as black, *i.e.*, pixels in color (0, 0, 0), and generate the colors for foreground categories using the pseudo-code elaborated in Figure S1.

During inference, to decode the output image to a single-channel ID map where each pixel represents a class ID, we compute the L1 distance between each output pixel and the pre-defined colors for each semantic category, and take the ID of the closest color as the predicted category. The pseudo-code for the post-processing is illustrated in Figure S2.

```
def generate_colors_dict(b, K):
    # b: the number of used values in a single channel
    # K: the number of classes

    m = 255 // b # get margin
    colors = []
    for class_id in range(K):
        # compute margin multiplier
        r_mult = class_id // b**2
        g_mult = (class_id % b**2) // b
        b_mult = class_id % b
        # compute r, g, b values
        r = 255 - r_mult * m
        g = 255 - g_mult * m
        b = 255 - b_mult * m
        colors.append((r, g, b))

    return colors
```

Figure S1. Pseudo-code for generating colors. //: floor division; %: mod operation.

```
def forward(image, colors):
    # image: (H, W, 3)
    # colors: (K, 3), where K is the number of classes

    # get distance between pixels and pre-defined colors
    dist = (image.view(H, W, 1, 3) - colors.view(1, 1, K, 3)
         ).abs().sum(-1) # (H, W, K)
    segm = dist.argmin(dim=-1) # (H, W)

    return segm
```

Figure S2. Pseudo-code of semantic segmentation post-processing.

Keypoint detection For keypoint detection, the output image consists of the R channel which denotes the class-agnostic heatmaps and the G/B channels that represent the keypoint categories. As illustrated in Figure S3, we convert the output image to a 17-channel heatmap, and follow the commonly used post-processing [39, 46] to obtain the final keypoint locations.

```
def forward(image, colors):
    # image: (H, W, 3)
    # colors: (K, 3), where K is the number of keypoints

    # r for heatmaps and gb for keypoint classes
    r = images[..., 0] # (H, W)
    gb = images[..., 1:] # (H, W, 2)

    # get keypoint class of each pixel
    dist = (gb.view(H, W, 1, 2) - colors.view(1, 1, K, 2)).
         abs().sum(-1) # (H, W, K)
    segm = dist.argmin(dim=-1) # (H, W)

    for idx in range(K):
        mask = segm == idx
        heatmap = mask * r # (H, W)
        heatmaps.append(heatmap)
    heatmaps = stack(heatmaps) # (K, H, W)

    return heatmaps
```

Figure S3. Pseudo-code of keypoint detection post-processing. `stack`: concatenates a sequence of tensors along a new dimension.

Panoptic segmentation As described in Section 3.2, we decompose the panoptic segmentation task into semantic segmentation and class-agnostic instance segmentation. During training, the semantic segmentation sub-task uses the same setting as the semantic segmentation on ADE-20K [54], except that we set the base $b = 7$ when assigning colors. Similar to the color generation process of semantic segmentation, we generate colors for each location category [42] used in class-agnostic instance segmentation. The color of each instance mask is determined by the location of its center.

During inference, the semantic ID map can be obtained using the post-processing described in Figure S2, while the class-agnostic instance masks are generated by thresholding the distance between predicted colors and the pre-defined colors for location categories. Matrix NMS [43] is adopted to remove duplicate instance predictions. We apply the majority vote of pixels from the semantic prediction to get the semantic class for each instance mask, as illustrated in Figure S4. Finally, we follow Panoptic FPN [26] to merge the semantic segmentation and the instance segmentation predictions to obtain the panoptic segmentation results.

Image restoration The detailed statistics of the datasets that are used for image restoration are shown in Table S1.

Tasks	Deraining							Enhance.	Denoising
Datasets	Rain14000 [15]	Rain1800 [49]	Rain800 [53]	Rain100H [49]	Rain100L [49]	Rain1200 [52]	Rain12 [29]	LoL [45]	SIDD [1]
Train Samples	11200	1800	700	0	0	0	12	485	320
Test Samples	2800	0	100	100	100	1200	0	15	40
Testset Rename	Test2800	-	Test100	Rain100H	Rain100L	Test1200	-	-	-

Table S1. Dataset description for image restoration tasks.

(a) Patch merging

merging?	mIoU	mAcc
✗	39.7	51.1
✓	41.2	53.0

(b) Encoder

backbone	mIoU	mAcc
ViT-B	31.4	41.4
ViT-L	41.2	53.0

(c) Head type

head	mIoU	mAcc
linear	38.6	50.2
light	41.2	53.0

(d) Loss function

loss	mIoU	mAcc
ℓ_1	40.6	52.5
ℓ_2	26.3	37.7
smooth-ℓ_1	41.2	53.0

Table S2. Ablation study on ADE-20K semantic segmentation. (a) merging patch after three transformer blocks; (b) encoder; (c) head type; (d) loss function.

```
def forward(segm_dist, inst_masks):
    # segm_dist: (H, W, K), where K is the number of thing
        classes
    # inst_masks: (N, H, W), where N is the number of
        instances

    # turn distances to scores
    segm_scores = 1. - semseg_dist / max(semseg_dist)
    # majority vote
    class_probs = einsum("nhw,hwk->nk", inst_masks,
        segm_scores) # (N, K)
    pred_classes = class_probs.argmax(dim=-1) # (N,)

    return pred_classes
```

Figure S4. Pseudo-code of majority voting for labeling each instance mask. `einsum`: Einstein summation.

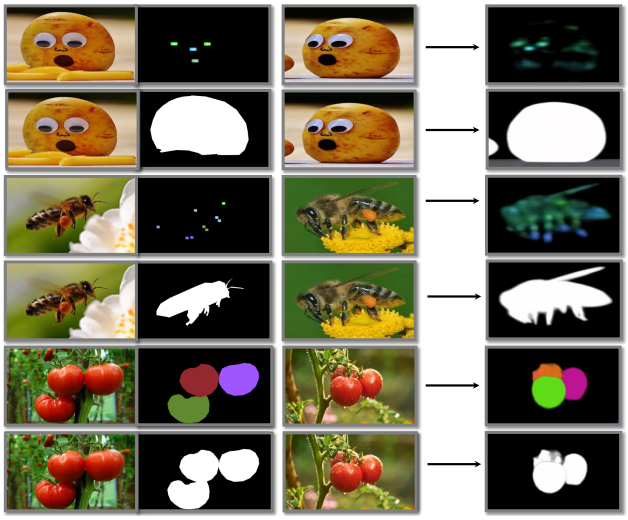

Figure S5. More visualizations. The visualized tasks include keypoint detection, object segmentation and instance segmentation on potato, bee, and tomato.

B. Additional Results

We report the results of ablation experiments on several components of our framework, with a shorter schedule of 3k iterations and other hyper-parameters unchanged on semantic segmentation of ADE-20K.

Merging patches During training, each input sample consists of both the input image and output image, which results in high memory cost and significantly slows down the training process. We reduce nearly half of the computation costs by merging the early features of the input image and the output image, *i.e.*, adding their features patch by patch after a three blocks. Table S2a shows that this new design even incurs performance increase. We argue that this design further provides the pixel-to-pixel correspondence between the input and its output via stacking them together. But in the original setting, these relationships need to be learned by the model, which will make the optimization more difficult especially in a short schedule.

Encoder We adopt standard Vision Transformer (ViT) [14] with different model sizes as the encoder, including ViT-base and ViT-large. Results are shown in Table S2b. We find that the model with ViT-L outperforms that with ViT-B by very large margins. This observation is intuitive, that generally larger models yield better performance. For generalist models, they can use more data but with less task-specific prior on method design, thus may require more model capacity than task-specific models.

Head We use a light three-layer head that consists of a linear (1×1 convolution) layer, a 3×3 convolution layer, and another linear layer, to map the feature of each patch to its original resolution, *e.g.*, $16 \times 16 \times 3$. The feature of

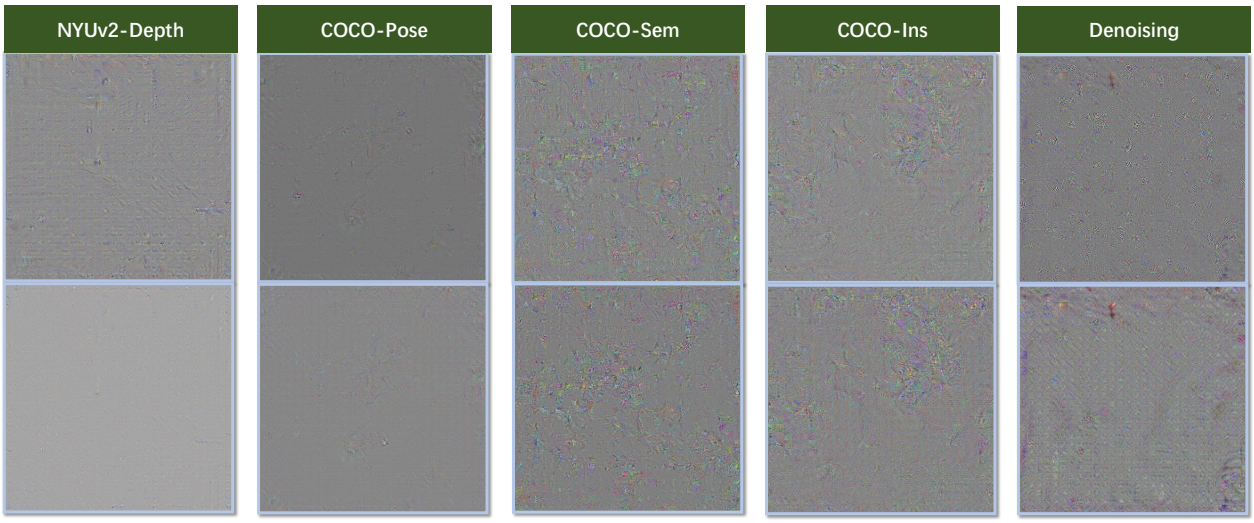

Figure S6. Visualizations of the learned prompts for different tasks. Each column denotes a prompt pair for a task. The first and second rows are input and output images respectively. For visualization, each prompt is normalized to an RGB image with values between 0 and 255. Different tasks show different patterns. Users can take the prompt images and feed them to the Painter model to enable the corresponding application.

each patch is the concatenation of the 4 feature maps evenly sampled from the transformer blocks. As shown in Table S2c, the light head achieves clear gains over the baseline with only a linear layer.

Loss function Painter uses a simple pixel regression loss to learn all the tasks. In Table S2d, we compare different regression loss functions, including ℓ_1, ℓ_2, and smooth-ℓ_1. We adopt smooth-ℓ_1 by default as it achieves the best performance and is also more stable during training.

C. Additional Visualization

In this section, we provide more visualizations. As shown in Figure S5, Painter performs in-context inference according to different prompt images. Note that Painter is never trained to solve these tasks during training, *e.g.*, keypoint detection of potato, object segmentation of bee, and instance segmentation of tomato.