A Implementation Details

Training Hyperparamters. RF-DETR extends LW-DETR (Chen et al., 2024a) for Neural Architecture Search. We highlight key differences in our training procedure below. First, we pseudo-label Objects365 (Shao et al., 2019) with SAM2 (Ravi et al., 2024) to allow us to pre-train the segmentation and detection heads on the same data. We use a learning rate of 1e-4 (LW-DETR uses 4e-4), and a batch size of 128 (LW-DETR uses the same). Similar to DINOv3 (Siméoni et al., 2025), we use an EMA scheduler since this is necessary for EMA's proper function. However, unlike DINOv3, we omit learning-rate warm-up. We clip all gradients greater than 0.1 and apply a per-layer multiplicative decay of 0.8 to preserve information (especially the earlier layers) in the DINOv2 backbone. We place our window attention blocks between layers $\{0, 1, 3, 4, 6, 7, 9, 10\}$, while LW-DETR places their window attention blocks between layers $\{0, 1, 3, 6, 7, 9\}$. Although we have the same number of windows, contiguous windowed blocks don't require an additional reshape operation, making our implementation slightly more efficient. Further, we train with more multi-scale resolutions (0.5 to 1.5 scale) than LW-DETR (0.7 to 1.4 scale) to ensure that the augmentation is symmetric around the default scale. Notably, we add resolution as a "tunable knob" in our NAS search space, while LW-DETR uses it as a form of data augmentation.

Latency Evaluation. We ensure fair evaluation between models by measuring detection accuracy and latency using the same artifact. To further standardize inference, we employ CUDA graphs in TensorRT, which pre-queue all kernels rather than requiring the CPU to launch them serially during execution. This optimization can accelerate some networks depending on the number and type of kernels used by the model. We observe that RT-DETR, LW-DETR, and RF-DETR benefit from this optimization. Further, CUDA graphs place LW-DETR on the same latency-accuracy curve as D-FINE, since CUDA graphs speed up LW-DETR but do not benefit D-FINE.

B Ablation on Query Tokens and Decoder Layers

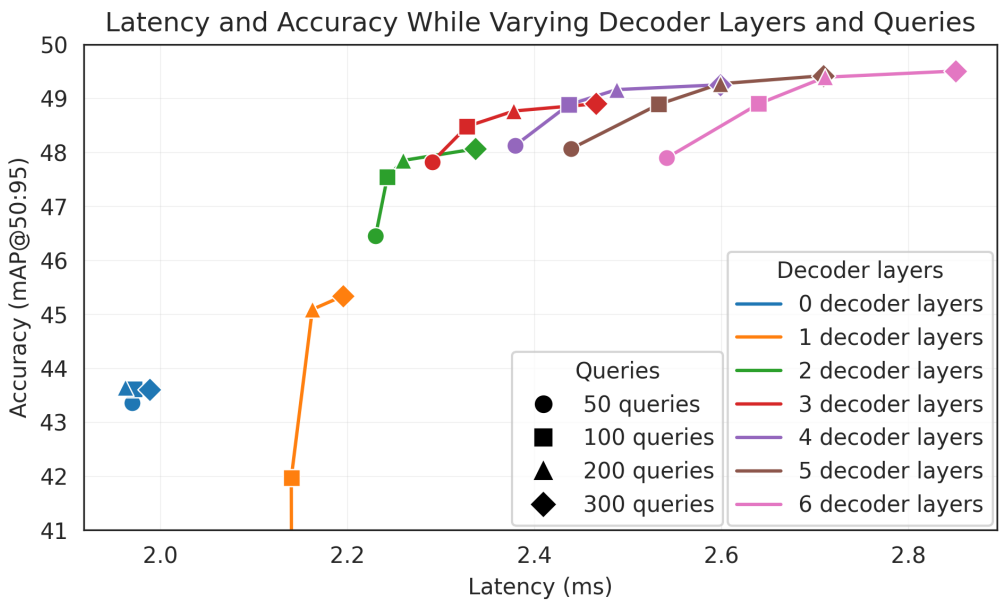

Figure 4: **Impact of Decoder Layers vs. Query Tokens**. We evaluate the impact of inference-time query dropping for trading-off accuracy and latency in RF-DETR (nano). Interestingly, we find that dropping the 100 lowest confidence queries does not significantly reduce performance, but modestly improves latency for all decoder layers.

We train RF-DETR (nano) with 300 object queries, following standard practice for real-time DETR-based object detectors. However, many datasets contain fewer than 300 objects per image. There-

fore, processing all 300 queries can be computationally wasteful. LW-DETR (tiny) demonstrates that training with fewer queries can improve the latency-accuracy tradeoff. Rather than deciding on the optimal number of queries apriori, we find that we can drop queries at test time *without retraining* by discarding the lowest-confidence queries ordered by the confidence of the corresponding token at the output of the encoder. As shown in Figure 4, this yields meaningful latency-accuracy tradeoffs. In addition, prior work (Zhao et al., 2024) demonstrates that decoder layers can be pruned at test time, since each layer is supervised independently during training. We find that it is possible to remove *all* decoder layers, relying solely on the initial query proposals from the two-stage DETR pipeline. In this case, there is no cross-attention to the encoder states or self-attention between queries, leading to a substantial runtime reduction. The resulting model resembles a single-stage YOLO-style architecture without NMS. As shown in Figure 4, eliminating the final decoder layer reduces latency by 10% with only a 2 mAP drop in performance.

C BENCHMARKING FLOPS

We benchmark FLOPs for RF-DETR GroundingDINO, and YOLO-E with PyTorch's `FlopCounterMode`. We find that `FlopCounterMode` closely reproduces FLOPs counts obtained with custom benchmarking tools for YOLOv11, D-FINE, and LW-DETR. In practice, we also find that it provides more reliable results than CalFLOPs (Ye, 2023). Notably, LW-DETR's FLOPs count is roughly twice that of the originally reported result (cf. Table 7). We posit that this discrepancy can be attributed to LW-DETR reporting FLOPs in FP16. We rely on the officially reported FLOPs counts from YOLOv11, YOLOv8, LW-DETR, and D-FINE.

Table 7: **FLOPs Benchmarking Comparison.** We compare FLOPs reported with custom benchmarking tools, CalFLOPs, and PyTorch's FlopCounterMode. Notably, we find that FlopCounterModel closely matches the results reported with custom benchmarking code, suggesting that it is more reliable than prior generic benchmarking tools.

Model	Size	Reported	CalFLOPs	FlopCounterMode
D-FINE	S	25.2 M	25.2 M	25.5 M
LW-DETR	S	16.6 M	22.9 M	31.8 M
YOLO11	S	21.5 M	23.9 M	21.6 M

D MODEL PREDICTIONS FROM RF-DETR AND RF-DETR-SEG

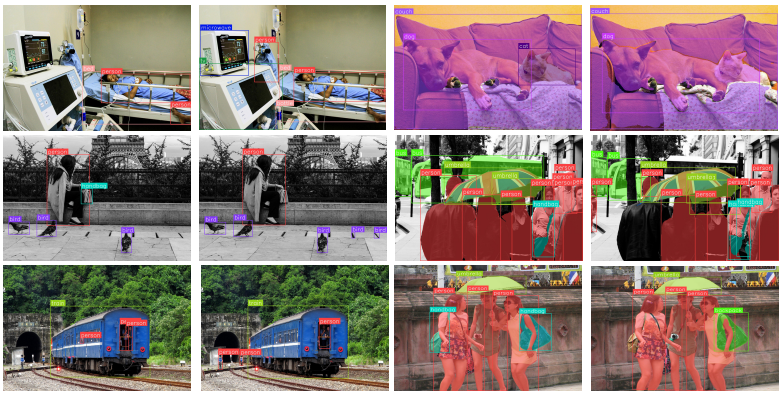

Figure 5: **Visualizing Model Predictions**. On the left, we compare detections from RF-DETR (nano) and LW-DETR (tiny). On the right, we compare instance segmentation masks from RF-DETR-Seg (nano) and YOLOv11 (nano)