

From Basics to Deployment

#### Technical Interview Preparation Guide

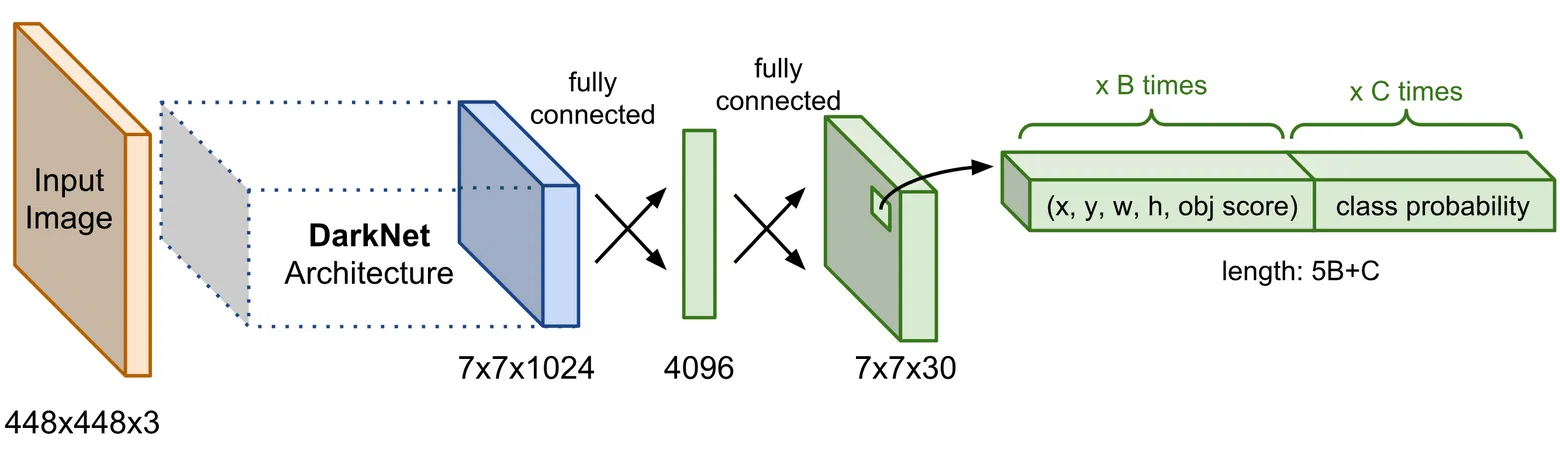
Computer Vision with YOLO: From Basics to Deployment

# YOLO Architecture & Working Principles

#### You Only Look Once: Single-Pass Object Detection

 **Single-Stage Detection:** Processes the entire image in one pass, predicting bounding boxes and class probabilities directly.

 **Grid-Based Approach:** Divides image into SxS grid cells, each responsible for detecting objects centered within it.



# YOLO prediction format

[x\_center, y\_center, width, height, confidence, class\_prob\_1, class\_prob\_2, ...]

# Single forward pass through the network predictions = model(image)

 **Anchor Boxes:** Predefined shapes that help better predict object dimensions and improve detection of overlapping

objects.

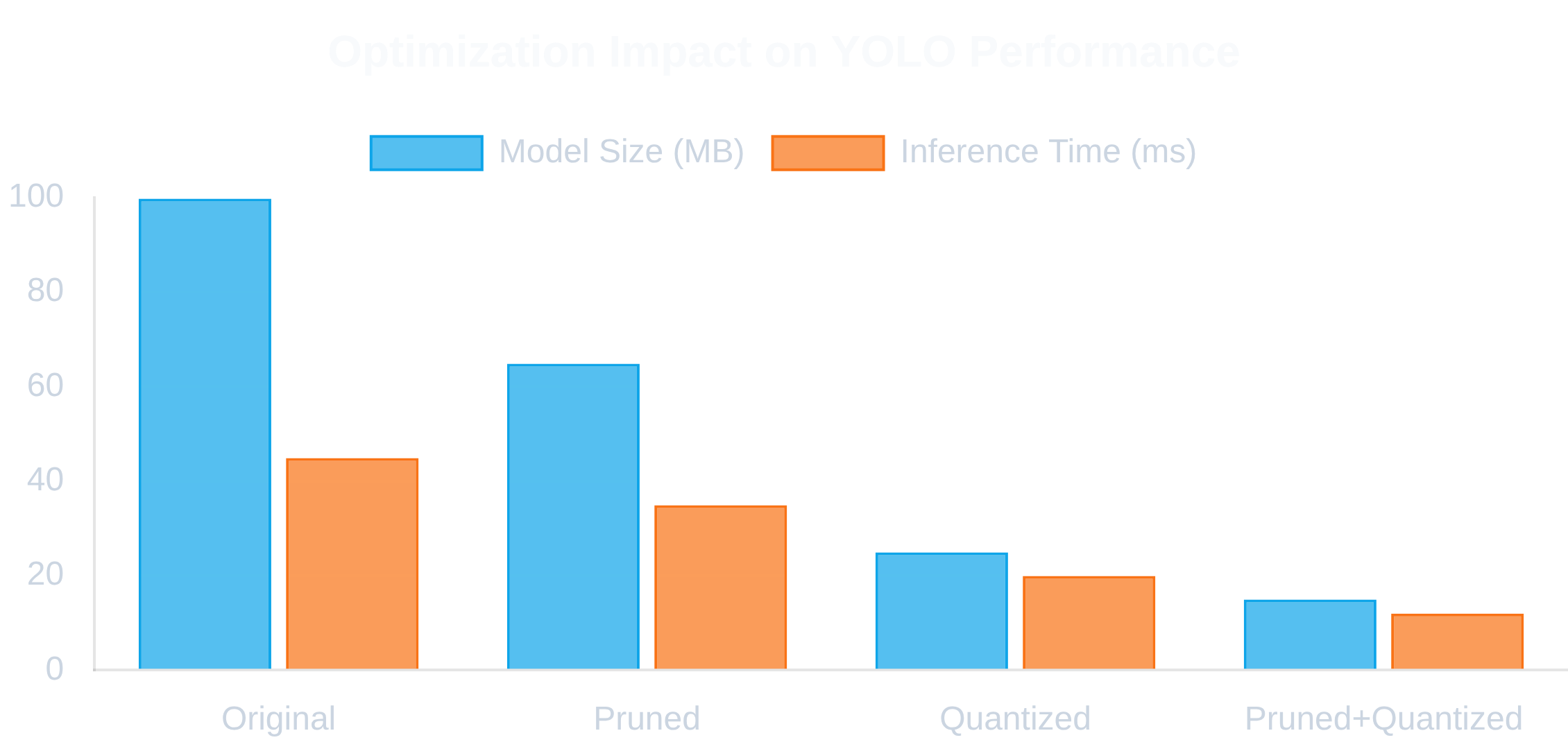
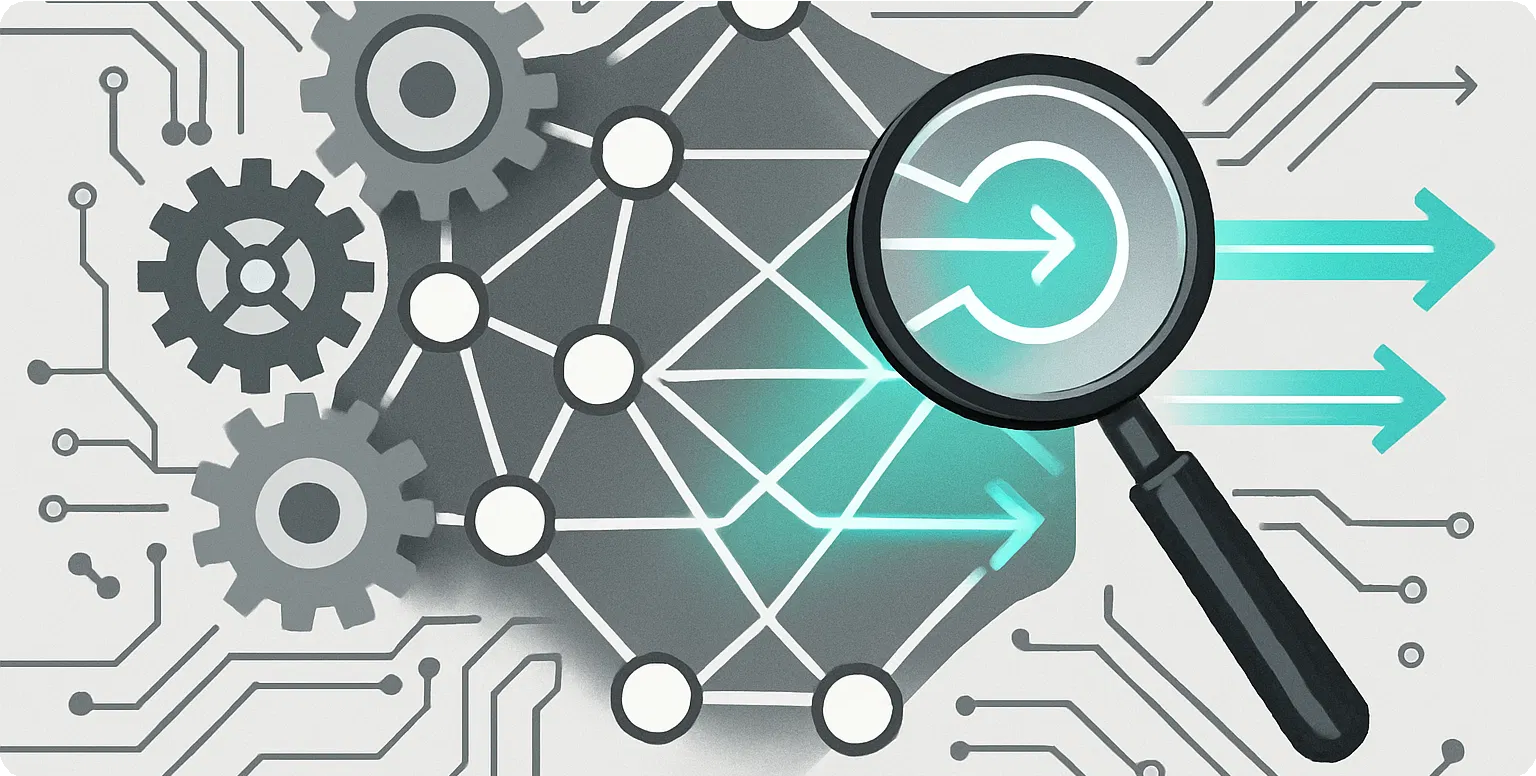
 **Key Components:** Backbone (feature extraction), Neck (feature aggregation), Head (detection).

 **Loss Function:** Combines localization loss (bounding box coordinates), confidence loss (objectness), and classification loss.

 **Non-Maximum Suppression:** Post- processing technique to eliminate redundant overlapping bounding boxes.

 **Speed vs. Accuracy:** Different model sizes (YOLOv5s, YOLOv5m, YOLOv5l, YOLOv5x) offer trade-offs between speed and accuracy.

Computer Vision with YOLO: From Basics to Deployment



# Optimization Techniques for YOLO

#### Enhancing Performance and Efficiency

 **Model Pruning:** Remove redundant connections and neurons to reduce model size without significant accuracy loss.

 **Quantization:** Convert floating-point weights to lower precision formats (INT8) to reduce memory footprint.

⚡ **Batch Processing:** Process multiple images simultaneously to better utilize GPU parallelism.

 **TensorRT Integration:** Convert models to TensorRT format for hardware-specific optimizations.

 **Input Resolution:** Find optimal balance between accuracy and speed by adjusting input size.

 **Deployment Optimization:** Tailor models for specific environments (AWS Lambda, edge devices).

Computer Vision with YOLO: From Basics to Deployment

# Transfer Learning with YOLO

#### Leveraging Pre-trained Models for New Tasks

 **Reduced Training Time:** Converges faster than training from scratch, often by 5-10x.

 **Less Data Required:** Achieve good results with smaller custom datasets (hundreds vs. thousands of images).

 **Better Performance:** Higher accuracy and lower overfitting on domain-specific tasks.

 **Lower Compute Needs:** Train effectively with less powerful hardware.

##### Start with Pre-trained Model

Begin with a YOLO model pre-trained on a large dataset like COCO.

##### Freeze Backbone Layers

Lock the weights of feature extraction layers to preserve learned features.

##### Modify Detection Head

Adapt the final layers to match your specific number of classes.

##### Fine-tune on Custom Data

Train with your domain-specific dataset using a lower learning rate.

##### Implementation Example:

# Load pre-trained YOLOv5 model

model = torch.hub.load('ultralytics/yolov5', 'yolov5s')

# Freeze backbone layers

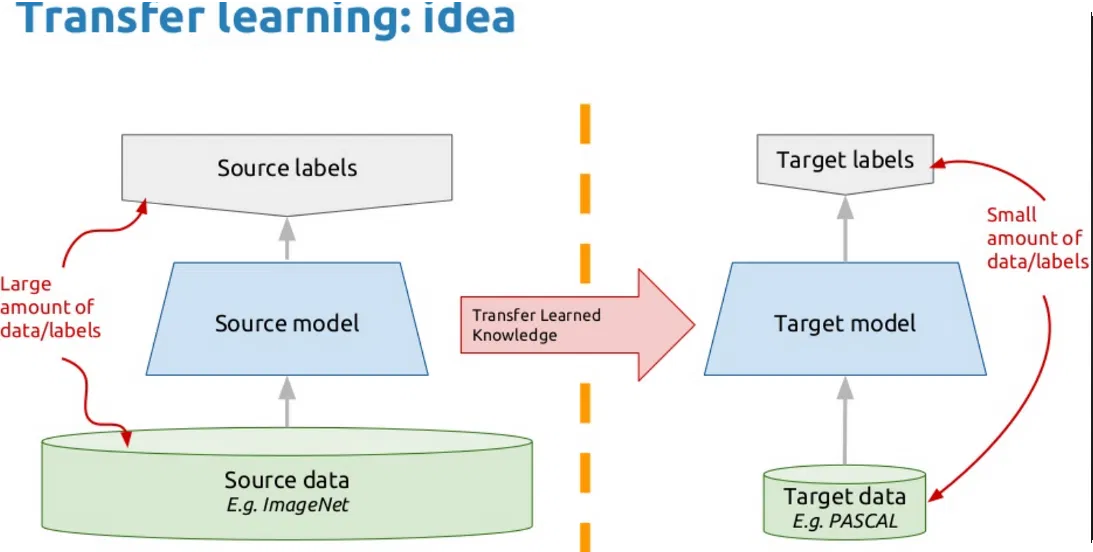
for param in model.model.model[:10].parameters(): param.requires\_grad = False

# Modify detection head for custom classes model.model.model[24].nc = num\_classes

# Fine-tune on custom dataset results = model.train(

data='custom.yaml', epochs=100, batch\_size=16, lr=0.001

)



Computer Vision with YOLO: From Basics to Deployment

# Structured JSON Output

#### Converting YOLO Outputs for Downstream Applications

{

"image\_name": "example\_image.jpg", "image\_dimensions": {

"width": 640,

"height": 480

},

"detections": [

{

"class\_name": "person", "class\_id": 0,

"confidence": 0.95,

"box\_2d": {

"x\_min": 100,

"y\_min": 50,

"x\_max": 200,

"y\_max": 300

}

},

{

"class\_name": "car",

 **Interoperability:** JSON is widely supported across programming languages and platforms.

 **Structured Data:** Hierarchical format provides clear organization of detection results.

 **Database Integration:** Easy to store in document databases like MongoDB.

 **API Friendly:** Perfect format for RESTful API responses.

# Python code to convert YOLO output to JSON

def yolo\_to\_json(detections, image\_name, image\_size): results = {

"image\_name": image\_name, "image\_dimensions": {

"width": image\_size[0], "height": image\_size[1]

},

"detections": []

}

for det in detections:

x, y, w, h, conf, cls = det results["detections"].append({

"class\_name": class\_names[int(cls)], "class\_id": int(cls),

"confidence": float(conf), "box\_2d": {

"x\_min": int((x - w/2) \* image\_size[0]),

"y\_min": int((y - h/2) \* image\_size[1]), "x\_max": int((x + w/2) \* image\_size[0]), "y\_max": int((y + h/2) \* image\_size[1])

}



})

results["detection\_count"] = len(results["detections"]) return results

Computer Vision with YOLO: From Basics to Deployment

# Building a Flask API for Real-time Analysis

#### Creating a Web Service for YOLO Object Detection

 **RESTful Endpoints:** Clean API design with proper HTTP methods and status codes.

 **Multiple Input Formats:** Support for file uploads, base64 encoded images, and URLs.

 **CORS Support:** Enable cross-origin requests for web applications.

 **Performance Optimization:** Load model once at startup for faster inference.

@app.route('/api/yolo/analyze', methods=['POST']) @cross\_origin()

def analyze\_image(): try:

 **Error Handling:** Robust error management with informative messages.

 **Health Checks:** Monitoring endpoints for system status.

# Check if image is provided

if 'image' not in request.files: return jsonify({

"error": "No image provided"

}), 400

# Process the image

file = request.files['image']

# Client-side JavaScript to call the API async function analyzeImage(imageFile) {

const formData = new FormData(); formData.append('image', imageFile);

try {

const response = await fetch( '[http://api.example.com/api/yolo/analyze',](http://api.example.com/api/yolo/analyze%27)

{

method: 'POST', body: formData



}

Computer Vision with YOLO: From Basics to Deployment

# Deploying YOLO on AWS Lambda

#### Serverless Computer Vision at Scale

AWS Lambda Architecture

##### Optimize Model

Prune and quantize the YOLO model to reduce size and improve inference speed.

**⚠ Size Limitations:** Lambda has 250MB package size limit. **Solution:** Use model quantization and container images.

**⚠ Cold Start Latency:** Initial invocation can be slow.

**Solution:** Implement provisioned concurrency and lazy loading.

**⚠ Memory Constraints:** Limited RAM for large models.

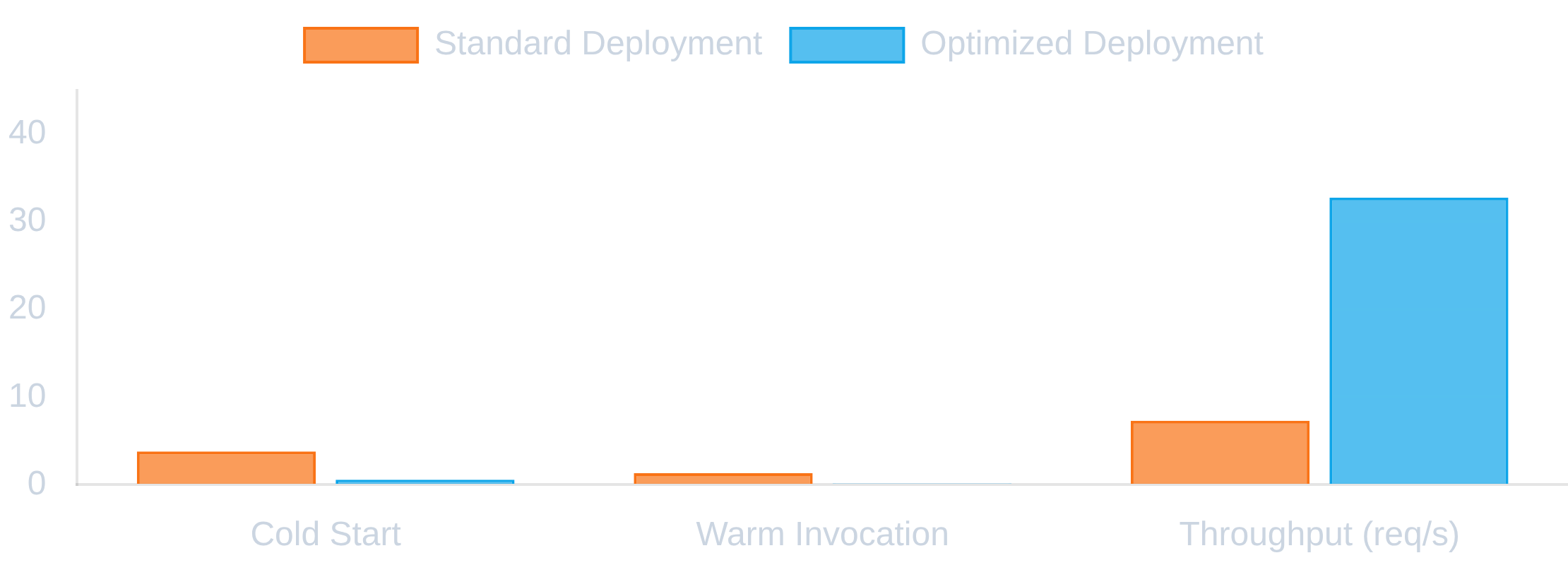
**Solution:** Increase allocated memory and optimize inference.

**⚠ Execution Timeout:** 15-minute maximum runtime.

**Solution:** Split processing pipeline into multiple functions.

Performance Metrics

##### Package Dependencies



Create deployment package with model weights and required libraries.

##### Configure API Gateway

Set up HTTP endpoints to trigger Lambda function.

##### Implement Monitoring

Add CloudWatch metrics and logs for performance tracking.

Computer Vision with YOLO: From Basics to Deployment

#### See the Complete System in Action

****

### Flask API Demo

Try the real-time object detection API with your own images.

**☁**

### AWS Lambda Demo

Test the serverless deployment with optimized performance.

**Launch Demo Launch Demo**

## 99.2%

Uptime

## 87.5%

[mAP@0.5](mailto:mAP@0.5)

## 45ms

Avg. Response Time

## 100+

Requests/Second



Thank you for your attention!