Production-Grade Deployment of RT-DETR: A Comprehensive Guide to Containerized, GPU-Accelerated Inference on WSL2

Section 1: Architecting the Foundation: WSL2 and NVIDIA GPU Integration

The successful deployment of a high-performance deep learning model hinges on a correctly configured and stable foundation. For this project, the foundation is a multi-layered architecture involving Windows 11, the Windows Subsystem for Linux (WSL), and direct access to a dedicated NVIDIA GPU. This section provides a meticulous, step-by-step guide to constructing this environment, ensuring that the powerful computational capabilities of the NVIDIA RTX 6000 Pro are seamlessly available within a containerized Linux environment.

1.1 The WSL2 GPU Passthrough Architecture: A Conceptual Overview

Before proceeding with installation, it is crucial to understand the underlying technology that enables GPU acceleration within WSL2. Unlike traditional virtual machines that require complex and often inefficient hardware passthrough configurations, WSL2 on modern Windows 11 systems utilizes a more elegant and performant approach integrated with the Windows Display Driver Model (WDDM).

This architecture operates on a client-server principle. The NVIDIA driver installed on the Windows 11 host system is the sole and complete driver package required. It contains not only the components for running native Windows applications but also a specialized kernel-mode driver and user-mode components designed to service requests from the WSL2 environment.

When a CUDA application runs inside the WSL2 Linux distribution, it does not interact with a native Linux driver controlling the physical hardware. Instead, it links against a special library, libcuda.so, which is stubbed into the WSL2 environment by the host driver installation. This library acts as a proxy, intercepting CUDA API calls made by the Linux application. These calls are then marshaled and forwarded across the WSL2 boundary to the host's WDDM driver, which executes the commands on the physical GPU and returns the results.

This tightly integrated model has profound implications for the setup process. It eliminates the need to install and maintain a separate, full-featured Linux display driver inside WSL2. In fact, attempting to do so would introduce a conflict, as two distinct driver stacks would compete for control of the same hardware, leading to instability or complete failure of GPU access from within WSL. Understanding this architecture transforms the setup from a sequence of arbitrary commands into a logical process, empowering more effective troubleshooting and system management.

1.2 Step-by-Step Host System Preparation

With the conceptual model established, the following steps will configure the host system. Each step must be completed and verified in sequence to ensure the integrity of the environment.

1.2.1 Windows 11 and WSL2 Installation

The first prerequisite is a modern Windows 11 installation and an up-to-date WSL2 kernel.

1. **Install WSL2:** Open a PowerShell or Command Prompt terminal with administrative privileges and execute the following command to install WSL and the default Ubuntu distribution:
   PowerShell
   `wsl.exe --install`

2. **Update the WSL Kernel:** It is imperative to ensure the WSL kernel is the latest version, as performance and compatibility improvements are frequently released. A kernel version of 5.10.16.3 or later is strongly recommended for optimal performance and functional fixes. Execute the following command:
   PowerShell
   `wsl.exe --update`

3. **Set WSL Version 2 as Default:** While the --install command typically defaults to version 2, it is good practice to confirm this setting:
   PowerShell
   wsl.exe --set-default-version 2

1.2.2 NVIDIA Driver Installation

Install the correct NVIDIA driver on the Windows 11 host. This is the most critical step for enabling GPU passthrough.

1. **Download the Driver:** Navigate to the official NVIDIA Driver Downloads page ((https://www.nvidia.com/Download/index.aspx)).
2. **Select Driver Details:**
   - **Product Type:** NVIDIA RTX / Quadro
   - **Product Series:** NVIDIA RTX Series (Notebooks/Desktops as appropriate)
   - **Product:** NVIDIA RTX 6000 Ada Generation (or your specific "RTX 6000 Pro" model)
   - **Operating System:** Windows 11
   - **Download Type:** NVIDIA RTX Enterprise / Production Branch (Recommended for stability) or Studio Driver (for creative applications).
3. **Install the Driver:** Run the downloaded installer and perform a "Clean Installation" if prompted to ensure no conflicting settings from previous drivers remain. Reboot the system after the installation is complete.

1.2.3 WSL Distribution Setup and Verification

With the host driver installed, the next step is to launch the Linux environment and verify GPU access.

1. **Launch WSL:** Open a new terminal and enter the WSL environment by typing:
   PowerShell
   wsl.exe

   This will launch the default Ubuntu distribution. It is advisable to update the package lists upon first launch:
   Bash
   sudo apt update && sudo apt upgrade -y

2. **Verify GPU Passthrough:** The definitive test of a successful setup is running the NVIDIA System Management Interface (nvidia-smi) tool from *within* the WSL terminal. The binary is located in a special path mapped from the host driver. Execute the following command:
   Bash
   `/usr/lib/wsl/lib/nvidia-smi`

   A successful execution will display a detailed report of the NVIDIA RTX 6000 Pro, including its name, driver version (which should match the host driver), and CUDA version. If this command fails or reports that no NVIDIA driver was found, revisit the previous installation steps.

1.3 Installing the Docker Ecosystem

The final layer of the foundation is the containerization platform. Docker Desktop for Windows provides seamless integration with the WSL2 backend, allowing Docker containers to leverage the same GPU passthrough mechanism.

1.3.1 Docker Desktop Installation

1. **Download and Install:** Download Docker Desktop for Windows from the official Docker website.
2. **Configure WSL2 Integration:** During the installation process, ensure the option "Use the WSL 2 based engine" is selected. This is the default and correct setting. If Docker Desktop is already installed, verify this setting in Settings > General.
3. **Enable for your Distribution:** Navigate to Settings > Resources > WSL Integration and ensure that integration is enabled for your installed Linux distribution (e.g., Ubuntu).

1.3.2 NVIDIA Container Toolkit Installation

The NVIDIA Container Toolkit is the essential bridge that allows the Docker daemon running within WSL2 to make containers GPU-aware.[1] It must be installed

inside the WSL2 Linux distribution.

1. **Enter WSL:** Launch a WSL terminal.

2. **Configure Package Repository:** Execute the following commands to add the NVIDIA container toolkit repository to your system's package manager:
   Bash
   ```
   curl -fsSL https://nvidia.github.io/libnvidia-container/gpgkey | sudo gpg --dearmor -o /usr/share/keyrings/nvidia-container-toolkit-keyring.gpg \
   && curl -s -L https://nvidia.github.io/libnvidia-container/stable/deb/nvidia-container-toolkit.list | \
   sed 's#deb https://#deb [signed-by=/usr/share/keyrings/nvidia-container-toolkit-keyring.gpg] https://#g' | \
   sudo tee /etc/apt/sources.list.d/nvidia-container-toolkit.list
   ```

3. **Install the Toolkit:** Update your package lists and install the toolkit:
   Bash
   ```
   sudo apt-get update
   sudo apt-get install -y nvidia-container-toolkit
   ```

4. **Restart Docker Daemon:** To ensure the Docker daemon recognizes the new runtime, it must be restarted. Since Docker Desktop manages the daemon, the simplest way is to restart Docker Desktop from the Windows system tray.

1.3.3 Final Ecosystem Verification

The final verification step confirms that all components—the host driver, WSL2, Docker, and the NVIDIA Container Toolkit—are working in concert.

1. **Run a CUDA Container:** From a WSL or PowerShell terminal, execute the following command:
   Bash
   ```
   docker run --rm --gpus all nvidia/cuda:12.1.1-base-ubuntu22.04 nvidia-smi
   ```

   This command pulls a basic NVIDIA CUDA image, runs a container, exposes all available GPUs (`--gpus all`), and executes `nvidia-smi` inside the container. A successful run will produce the same GPU report seen in step 1.2.3, but this time from within an isolated Docker container. This confirms the entire stack is ready for the RT-DETR application deployment.

Section 2: The Inference Engine: Optimizing RT-DETR

with TensorRT

With the foundational infrastructure in place, the focus now shifts to the core asset: the RT-DETR model itself. This section outlines the process of selecting the most appropriate model implementation and pre-trained weights, followed by a critical optimization pipeline to convert the model from its native PyTorch format into a highly efficient TensorRT engine, purpose-built for the target hardware.

2.1 Choosing Your RT-DETR Implementation: Official vs. Ultralytics

Baidu's RT-DETR, a state-of-the-art Real-Time Detection Transformer, has demonstrated performance that rivals and, in some cases, surpasses the popular YOLO series of models.[5] The model is available through two primary channels:

1. **Official lyuwenyu/RT-DETR Repository:** This GitHub repository is the original source, containing implementations in both PaddlePaddle and PyTorch. It is the definitive reference for the model's architecture and research.
2. **Ultralytics Framework Integration:** The Ultralytics framework, well-known for its YOLO implementations, has integrated RT-DETR into its ecosystem.[7] This integration provides a high-level, unified API for training, validation, inference, and, most importantly for this guide, model exporting.[9]

While the official repository is invaluable for research, the path from its PyTorch code to a production-ready deployment is fraught with complexity. Manual conversion from PyTorch to ONNX and then to TensorRT often involves troubleshooting unsupported operations, simplifying the computational graph with external tools, and managing dynamic input shapes.[10]

The Ultralytics framework abstracts away this entire toolchain behind a simple, robust API. For the goal of creating a deployable inference service, leveraging the Ultralytics implementation is the most efficient and reliable path. It represents a strategic choice to utilize a mature MLOps framework that prioritizes ease of deployment and reproducibility, reflecting a broader industry trend of abstracting complex deployment pipelines. Therefore, this guide will proceed using the Ultralytics implementation.

2.2 Model Selection: Balancing Speed and Accuracy

The RT-DETR family includes several models with varying backbones and sizes, offering a trade-off between inference speed and detection accuracy. The choice of model depends on the specific requirements of the application. The table below, compiled from performance benchmarks, summarizes the key characteristics of the larger, more accurate RT-DETR variants.[5]

Model Name	Backbone	COCO mAP (val)	Parameters (M)	FLOPs (G)	T4 TensorRT FP16 FPS
RT-DETR-L	HGNetv2-L	53.0	32	110	114
RT-DETR-X	HGNetv2-X	54.8	67	234	74
RT-DETRv2-L	ResNet-50	53.4	42	136	108
RT-DETRv2-X	ResNet-101	54.3	76	259	74

For this guide, the rtdetr-l.pt model will be used. It offers a compelling balance, providing high accuracy (53.0 mAP) with excellent throughput (114 FPS on a T4 GPU), making it a suitable and representative choice for a general-purpose object detection API.[7]

2.3 The Optimization Pipeline: PyTorch → ONNX → TensorRT

The goal of this pipeline is to transform the flexible, training-oriented PyTorch model into a rigid, highly optimized inference engine. This is a three-step process.

2.3.1 Environment Setup

Create a dedicated Python environment within your WSL2 distribution to handle the

conversion.

1. **Create a Project Directory:**
   Bash
   ```
   mkdir ~/rtdetr_conversion
   cd ~/rtdetr_conversion
   ```

2. **Install Dependencies:** Create a file named requirements_export.txt with the following content:
   ```
   ultralytics
   torch torchvision torchaudio --index-url https://download.pytorch.org/whl/cu121
   onnx
   onnxsim
   ```

   Install these packages:
   Bash
   ```
   python3 -m venv venv
   source venv/bin/activate
   pip install -r requirements_export.txt
   ```

2.3.2 Step 1: Export to ONNX

The Open Neural Network Exchange (ONNX) is an open standard for representing machine learning models. It serves as an intermediate representation that decouples the model from its original framework, enabling it to be used with a variety of inference engines.[11]

1. **Create an Export Script:** Create a Python script named export_onnx.py:
   Python
   ```
   from ultralytics import RTDETR

   # Load the pre-trained RT-DETR-L model
   model = RTDETR('rtdetr-l.pt')

   # Export the model to ONNX format with FP16 precision
   # This will create a file named 'rtdetr-l.onnx'
   model.export(format='onnx', half=True)

   print("Model exported to rtdetr-l.onnx")
   ```

2. **Run the Script:**

```
Bash
python export_onnx.py
```

This will download the rtdetr-l.pt weights if not already present and generate an rtdetr-l.onnx file in the current directory. The half=True argument exports the model weights in 16-bit floating-point format, which is ideal for TensorRT's mixed-precision capabilities.

2.3.3 Step 2: Convert ONNX to a TensorRT Engine

NVIDIA TensorRT is a high-performance deep learning inference optimizer and runtime. It takes a trained model (in this case, via ONNX) and performs numerous optimizations, including layer and tensor fusion, kernel auto-tuning, and precision calibration, specifically for the target GPU architecture.[8] The output is a self-contained, serialized file called an "engine" that is maximally optimized for inference on that specific hardware.

1. **Locate trtexec:** The TensorRT installation that comes with the NVIDIA base Docker images (or can be installed separately) includes a powerful command-line tool called trtexec. This tool is used to build engines from ONNX models and benchmark their performance.
2. **Execute the Conversion:** Run the following command in your WSL terminal. This command must be run on a system with TensorRT installed. The easiest way is to use the same Docker container that will eventually run the service.

   ```
   Bash
   # First, launch a container with the project directory mounted
   docker run --rm -it --gpus 'device=1' -v $(pwd):/workspace
   nvcr.io/nvidia/pytorch:25.08-py3 bash

   # Inside the container, navigate to the workspace and run trtexec
   cd /workspace
   /usr/src/tensorrt/bin/trtexec --onnx=rtdetr-l.onnx \
                         --saveEngine=rtdetr-l.engine \
                         --fp16
   ```

   - --onnx=rtdetr-l.onnx: Specifies the input ONNX model file.
   - --saveEngine=rtdetr-l.engine: Specifies the output path for the compiled TensorRT engine.
   - --fp16: Enables fast FP16 (half-precision) mode for inference, which offers a significant speedup with minimal to no loss in accuracy on supported GPUs like the RTX 6000 Pro.[11]

Upon completion, you will have a file named rtdetr-l.engine. This file is the final, optimized artifact that will be deployed within the API service. It is portable but hardware-specific; it is optimized for the GPU it was built on and may not perform optimally (or run at all) on a different GPU architecture.

Section 3: Service Implementation: Building the Detection API with FastAPI

With the optimized TensorRT engine ready, the next stage is to build the web service that will load this engine, accept image data via an HTTP endpoint, perform inference, and return structured detection results. FastAPI is an ideal framework for this task due to its high performance, asynchronous capabilities, and automatic data validation and API documentation generation.[14]

3.1 Project Structure and Dependencies

A well-organized project structure is essential for maintainability and scalability. The service will be structured as follows:

```
/rtdetr_api
├── app/
│   ├── __init__.py
│   ├── main.py         # FastAPI application, routes, and logic
│   └── inference.py    # TensorRT engine wrapper and processing logic
├── models/
│   └── rtdetr-l.engine # The compiled TensorRT engine from Section 2
├── requirements.txt
└── Dockerfile
```

The requirements.txt file will contain all necessary Python dependencies for the application to run:

```
fastapi
uvicorn[standard]
python-multipart
numpy
opencv-python-headless
# TensorRT is provided by the base Docker image, but for local dev you would install it here
# tensorrt
```

- fastapi: The core web framework.
- uvicorn: The ASGI server that runs the FastAPI application.
- python-multipart: Required for handling file uploads.
- numpy: For numerical operations on tensors and image data.
- opencv-python-headless: For image decoding and preprocessing, without GUI dependencies.

3.2 The Inference Engine Wrapper (inference.py)

This module is the heart of the application's machine learning capability. It encapsulates all interactions with the TensorRT engine, abstracting the low-level details of memory management and execution from the web framework logic. This separation of concerns follows best practices for building robust ML services.[16]

Python

```
# app/inference.py

import tensorrt as trt
import numpy as np
import cv2
import torch
from typing import List, Dict, Any, Tuple


def _trt_dtype_to_torch(dtype: trt.DataType) -> torch.dtype:
```

```
    """Map TensorRT data types to PyTorch dtypes."""
    mapping = {
        trt.DataType.FLOAT: torch.float32,
        trt.DataType.HALF: torch.float16,
        trt.DataType.INT8: torch.int8,
        trt.DataType.INT32: torch.int32,
        trt.DataType.BOOL: torch.bool,
    }
    if dtype not in mapping:
        raise TypeError(f"Unsupported TensorRT dtype: {dtype}")
    return mapping[dtype]


def _dims_to_tuple(dims: trt.Dims) -> Tuple[int, ...]:
    """Convert TensorRT Dims to a Python tuple."""
    return tuple(int(d) for d in dims)


class RTDETREngine:
    """
    Minimal TensorRT inference wrapper that uses PyTorch tensors for GPU buffers.

    Assumes a single input binding and one or more outputs.
    Postprocessing assumes RT-DETR-like output of shape [1, num_queries, 6] with
    per-query [cx, cy, w, h, class_id, score] where coords are normalized to [0,1].
    Adjust _postprocess() if your export is different.
    """
    def __init__(self, engine_path: str, conf_threshold: float = 0.5):
        self.logger = trt.Logger(trt.Logger.WARNING)
        trt.init_libnvinfer_plugins(self.logger, "")

        try:
            with open(engine_path, "rb") as f, trt.Runtime(self.logger) as runtime:
                self.engine = runtime.deserialize_cuda_engine(f.read())
        except Exception as e:
            raise RuntimeError(f"Error loading TensorRT engine: {e}")

        if self.engine is None:
            raise RuntimeError("Failed to deserialize TensorRT engine.")

        self.context = self.engine.create_execution_context()
        if self.context is None:
            raise RuntimeError("Failed to create TensorRT execution context.")

        # Detect API flavor
        self.use_v3 = hasattr(self.engine, "num_io_tensors")  # TRT 9/10+
        self.conf_threshold = conf_threshold
```

```
        if self.use_v3:
            # New API: work with tensor names
            self.all_tensor_names = [self.engine.get_tensor_name(i) for i in
range(self.engine.num_io_tensors)]
            self.input_names = [
                n for n in self.all_tensor_names
                if self.engine.get_tensor_mode(n) == trt.TensorIOMode.INPUT
            ]
            self.output_names = [
                n for n in self.all_tensor_names
                if self.engine.get_tensor_mode(n) == trt.TensorIOMode.OUTPUT
            ]
            if len(self.input_names) != 1:
                raise ValueError(f"Expected exactly 1 input, found: {self.input_names}")
            self.output_tensors: Dict[str, Tuple[torch.Tensor, Tuple[int, ...]]] = {}
        else:
            # Legacy API: indices & bindings
            if not hasattr(self.engine, "num_bindings"):
                raise RuntimeError("Neither v3 tensors API nor legacy num_bindings found.")
            self.num_bindings = self.engine.num_bindings
            self.input_indices = [i for i in range(self.num_bindings) if self.engine.binding_is_input(i)]
            self.output_indices = [i for i in range(self.num_bindings) if not self.engine.binding_is_input(i)]
            if len(self.input_indices) != 1:
                raise ValueError(f"Expected exactly 1 input, got {len(self.input_indices)}.")
            self.bindings: List[int] = [0] * self.num_bindings
            self.output_tensors_legacy: Dict[int, Tuple[torch.Tensor, Tuple[int, ...]]] = {}

    # ---------- Pre/Post ----------
    def _preprocess(self, image: np.ndarray) -> torch.Tensor:
        """Resize to 640x640, BGR->RGB, normalize to [0,1], HWC->NCHW, add batch, send to CUDA."""
        img = cv2.resize(image, (640, 640))
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        img = img.astype(np.float32) / 255.0
        img = np.transpose(img, (2, 0, 1))  # CHW
        tensor = torch.from_numpy(img).unsqueeze(0).contiguous()  # [1,3,640,640]
        return tensor.cuda(non_blocking=True)

    def _postprocess(self, output: torch.Tensor, original_hw: Tuple[int, int]) -> List[Dict[str, Any]]:
        """
        Parse RT-DETR-like output.       Accepts shapes:
          - [1, N, 6]  -> (cx,cy,w,h,class_id,score)
          - [1, 6, N]  -> same, transposed
        Returns a list of dicts with [x1,y1,x2,y2] in pixel coords of the original image.
        """
        h, w = original_hw
        detections: List[Dict[str, Any]] = []

        if output.ndim != 3:
```

```
            raise ValueError(f"Unexpected output ndim: {output.ndim}, expected 3 (e.g., [1, N, 6]).")

        if output.shape[2] == 6:
            # [1, N, 6]
            dets = output[0]  # [N, 6]
        elif output.shape[1] == 6:
            # [1, 6, N] -> transpose to [N, 6]
            dets = output[0].transpose(0, 1)  # [N, 6]
        else:
            raise ValueError(f"Unexpected output shape: {tuple(output.shape)}; cannot infer last-dim=6.")

        # Ensure float32 on CPU for Python-side math
        dets = dets.float().detach().cpu().numpy()

        for det in dets:
            cx, cy, bw, bh, cls_id, score = det.tolist()
            if score < self.conf_threshold:
                continue

            # Convert from center format (normalized) to xyxy pixel coords
            x1 = (cx - bw / 2.0) * w
            y1 = (cy - bh / 2.0) * h
            x2 = (cx + bw / 2.0) * w
            y2 = (cy + bh / 2.0) * h

            # Clamp to image bounds
            x1 = float(np.clip(x1, 0, w - 1))
            y1 = float(np.clip(y1, 0, h - 1))
            x2 = float(np.clip(x2, 0, w - 1))
            y2 = float(np.clip(y2, 0, h - 1))

            detections.append({
                "box": [x1, y1, x2, y2],
                "score": float(score),
                "label": int(cls_id),
            })

        return detections

    # ---------- Inference ----------
    def _setup_bindings(self, input_tensor: torch.Tensor) -> None:
        """
        Set dynamic shapes and allocate output buffers on CUDA.
        Uses synchronous execute_v2 for simplicity/robustness.
        """
        assert input_tensor.is_cuda, "Input tensor must be on CUDA."

        inp_idx = self.input_indices[0]
```

```
        # Set the input shape, e.g., [1,3,640,640]
        self.context.set_binding_shape(inp_idx, tuple(input_tensor.shape))

        # Prepare bindings list with correct pointers
        self.bindings = [0] * self.num_bindings
        self.bindings[inp_idx] = int(input_tensor.data_ptr())

        # Allocate outputs
        self.output_tensors = {}
        for out_idx in self.output_indices:
            out_dtype = _trt_dtype_to_torch(self.engine.get_binding_dtype(out_idx))
            out_shape = _dims_to_tuple(self.context.get_binding_shape(out_idx))
            if any(d < 0 for d in out_shape):
                raise RuntimeError(f"Unresolved dynamic output shape for binding {out_idx}: {out_shape}")
            # Allocate as flat and reshape later
            numel = int(np.prod(out_shape)) if len(out_shape) > 0 else 1
            out_buf = torch.empty(numel, dtype=out_dtype, device='cuda')
            self.bindings[out_idx] = int(out_buf.data_ptr())
            self.output_tensors[out_idx] = (out_buf, out_shape)

    def _setup_v3_tensors(self, input_tensor: torch.Tensor) -> None:
        inp_name = self.input_names[0]
        # Set input shape and address
        self.context.set_input_shape(inp_name, tuple(input_tensor.shape))
        self.context.set_tensor_address(inp_name, int(input_tensor.data_ptr()))

        # Allocate outputs after shapes are known
        self.output_tensors = {}
        for name in self.output_names:
            out_dtype = _trt_dtype_to_torch(self.engine.get_tensor_dtype(name))
            out_shape = _dims_to_tuple(self.context.get_tensor_shape(name))
            if any(d < 0 for d in out_shape):
                # Some engines need shape inference; try to proceed—TRT resolves before execute_v3.
                # We'll allocate pessimistically on first run if needed (rare).
                raise RuntimeError(f"Unresolved dynamic output shape for tensor '{name}': {out_shape}")
            numel = int(np.prod(out_shape)) if len(out_shape) > 0 else 1
            out_buf = torch.empty(numel, dtype=out_dtype, device='cuda')
            self.context.set_tensor_address(name, int(out_buf.data_ptr()))
            self.output_tensors[name] = (out_buf, out_shape)


    def detect(self, image: np.ndarray) -> List[Dict[str, Any]]:
        original_hw = image.shape[:2]  # (h, w)
        input_tensor = self._preprocess(image)

        # Hook up engine bindings
        self._setup_bindings(input_tensor)
```

```
        # Execute synchronously
        ok = self.context.execute_v2(self.bindings)
        if not ok:
            raise RuntimeError("TensorRT execution failed.")

        # Gather outputs (assume first output contains detections)
        # If multiple outputs, you may need to adapt this selection.
        first_out_idx = self.output_indices[0]
        out_buf, out_shape = self.output_tensors[first_out_idx]
        output = out_buf.reshape(out_shape)

        # Post-process
        results = self._postprocess(output, original_hw)
        return results
```

3.3 The FastAPI Application (main.py)

This script sets up the web server, defines the API endpoints, and orchestrates the inference process using the RTDETREngine class.

A critical design choice is made in defining the prediction endpoint. While FastAPI is an asynchronous framework, TensorRT inference is a synchronous, blocking, and computationally intensive operation. If this blocking call were placed inside an async def route, it would halt the server's main event loop, preventing it from handling any other requests until the inference completes. This would effectively serialize all requests and destroy the concurrency benefits of the framework.

To avoid this, the endpoint is defined as a standard synchronous function (def). FastAPI is intelligent enough to run such functions in a separate thread pool, which keeps the main event loop free to accept new incoming requests while the heavy computation happens in the background. This is the correct and robust pattern for serving synchronous ML models with an asynchronous web framework.[14]

Python

```python
# app/main.py

from fastapi import FastAPI, UploadFile, File, HTTPException
from pydantic import BaseModel, Field
from typing import List
import numpy as np
import cv2
import io

from.inference import RTDETREngine

# --- Pydantic Models for Structured I/O ---
class DetectionBox(BaseModel):
    box: List[float] = Field(..., example=[100.0, 150.0, 200.0, 250.0])
    score: float = Field(..., example=0.95)
    label: int = Field(..., example=17) # COCO class ID for 'cat'

class DetectionResponse(BaseModel):
    detections: List

# --- FastAPI Application ---
app = FastAPI(
    title="RT-DETR Inference API",
    description="An API to perform object detection using a TensorRT-optimized RT-DETR model.",
    version="1.0.0"
)

# Load the model on startup. This is a global instance.
# The model is loaded only once, which is crucial for performance.
try:
    engine = RTDETREngine(engine_path="/app/models/rtdetr-l.engine")
except Exception as e:
    # If the model fails to load, the app is not functional.
    # A more robust solution might involve a health check endpoint.
    print(f"Error loading TensorRT engine: {e}")
    engine = None

@app.on_event("startup")
async def startup_event():
    if engine is None:
        raise RuntimeError("TensorRT engine could not be loaded. The application cannot start.")
    print("RT-DETR TensorRT Engine loaded successfully.")
```

```python
@app.get("/health", summary="Check if the API is running")
def health_check():
    return {"status": "ok"}

@app.post("/predict", response_model=DetectionResponse, summary="Perform Object Detection")
def predict(file: UploadFile = File(...)):
    if not file.content_type.startswith("image/"):
        raise HTTPException(status_code=400, detail="File provided is not an image.")

    try:
        # Read image bytes and decode with OpenCV
        contents = file.file.read()
        image_array = np.frombuffer(contents, np.uint8)
        image = cv2.imdecode(image_array, cv2.IMREAD_COLOR)

        if image is None:
            raise HTTPException(status_code=400, detail="Could not decode image.")

        # Perform inference
        detections = engine.detect(image)

        return {"detections": detections}

    except Exception as e:
        # Log the error for debugging
        print(f"An error occurred during prediction: {e}")
        raise HTTPException(status_code=500, detail="An internal error occurred during processing.")
```

Section 4: Encapsulation: Containerizing the Service with Docker

The final step in creating a portable and reproducible deployment artifact is to encapsulate the entire application—including its code, dependencies, and the model engine—into a Docker image. This process ensures that the service runs identically regardless of the underlying host environment, provided the necessary NVIDIA driver and container toolkit are present.

4.1 Choosing the Right Base Image

The choice of the base Docker image is the single most critical decision in this process. Attempting to build a GPU-enabled application starting from a generic base image like python:3.10-slim [17] would require manually installing the correct versions of the CUDA toolkit, cuDNN, and TensorRT, a notoriously complex and error-prone task.

The correct approach is to leverage the official, pre-built container images provided by NVIDIA on their NGC (NVIDIA GPU Cloud) registry. These images are meticulously maintained and come with the entire NVIDIA software stack pre-installed and validated.[18] For this project, an image such as

nvcr.io/nvidia/pytorch:23.10-py3 is an excellent choice. It includes a compatible version of Python, PyTorch (useful for tensor manipulations), CUDA, cuDNN, and, most importantly, TensorRT.[20] This choice drastically reduces setup complexity and eliminates a major source of potential deployment failures.

4.2 Constructing the Dockerfile

The Dockerfile is a text file that contains the instructions for building the Docker image. It specifies the base image, copies the application files, installs dependencies, and defines the command to run when a container is started.

For production environments, a multi-stage build is the recommended best practice. This technique involves using a larger "builder" image that contains development tools (like compilers) to create assets, and then copying only the necessary runtime artifacts into a smaller, final image. This minimizes the size and attack surface of the production container. While a single-stage build is sufficient for this guide's purpose, understanding the multi-stage pattern is key for maturing the deployment process.

The following single-stage Dockerfile will be used to build the service:

Dockerfile

```
# Dockerfile
```

```
FROM nvcr.io/nvidia/pytorch:25.08-py3

# Work in a stable path
WORKDIR /opt/app

# Create non-root user (we'll switch to it after installs)
RUN useradd -ms /bin/bash appuser && chown -R appuser:appuser /opt/app

# Copy only requirements first for better layer caching
COPY requirements.txt /tmp/requirements.txt

# Install Python deps as root
RUN pip install --no-cache-dir -r /tmp/requirements.txt && rm -rf /root/.cache/pip

RUN pip install --no-cache-dir 'numpy<2.0'

RUN ln -s /opt/app /app

# now switch to appuser and copy sources (your existing lines)
USER appuser
COPY --chown=appuser:appuser app/ /opt/app/app/
COPY --chown=appuser:appuser models/ /opt/app/models/


# Now switch to non-root and copy app sources
USER appuser

WORKDIR /opt/app

#  Note the spaces and correct src/dst
COPY --chown=appuser:appuser app ./app
COPY --chown=appuser:appuser models ./models

EXPOSE 8000

CMD ["uvicorn", "app.main:app", "--host", "0.0.0.0", "--port", "8000"]
```

A .dockerignore file should also be created in the project root to prevent unnecessary files (like virtual environments and caches) from being copied into the image, which speeds up the build process.

```
#.dockerignore
```

```
__pycache__/
*.pyc
*.pyo
*.pyd
.Python
env/
venv/
.git
.idea
```

Section 5: Deployment and End-to-End Verification

This final section integrates all previous efforts. It provides the commands to build the Docker image, run the containerized service, and, most importantly, verify its functionality with an external client script. This end-to-end test provides definitive proof that the entire technology stack is operating correctly.

5.1 Building and Running the Container

These commands should be executed from the project root directory (/rtdetr_api) within the WSL2 terminal.

1. **Build the Docker Image:** This command reads the Dockerfile, executes its instructions, and creates a local Docker image tagged as rtdetr-api.
   Bash
   ```
   docker build -t rtdetr-api.
   ```

2. **Run the Docker Container:** This command starts a new container from the rtdetr-api image and connects it to the host system's resources.
   Bash
   ```
   docker run --rm -it --gpus all -p 8000:8000 rtdetr-api
   ```

   A breakdown of the flags:
   - --rm: Automatically removes the container when it exits, which is useful for cleanup during development.
   - -it: Runs the container in interactive mode and allocates a pseudo-TTY, allowing you to see the application logs (e.g., Uvicorn startup messages) in your terminal.

- --gpus all: This is the crucial flag that instructs the Docker daemon to use the NVIDIA Container Toolkit to expose the host's GPU(s) to the container.
- -p 8000:8000: This maps port 8000 on the host machine to port 8000 inside the container, making the API accessible from the host at http://localhost:8000.

Upon successful execution, you should see logs from Uvicorn indicating that the application has started and is listening for connections on 0.0.0.0:8000.

5.2 Verification with a Client Script

The ultimate validation is to interact with the API as an external client, fulfilling the user's core requirement to "hit it as an api from other local instances." The following Python script (test_client.py) simulates this interaction. It should be run from a separate terminal on the host machine (either WSL or Windows, as long as Python and requests are installed).

Create a file named test_client.py in a directory outside the API project, and place a test image (e.g., my_image.jpg) in the same directory.

Python

```
# test_client.py

import requests
import json
from pprint import pprint

# The URL of the API endpoint
API_URL = "http://localhost:8000/predict"

# The path to the image you want to test
IMAGE_PATH = "my_image.jpg"

def test_detection_api(image_path):
    """
    Sends an image to the detection API and prints the response.
    """
    try:
        with open(image_path, "rb") as image_file:
```

```
            files = {"file": (image_path, image_file, "image/jpeg")}

            print(f"Sending request for image: {image_path}")
            response = requests.post(API_URL, files=files)

            # Check for successful response
            response.raise_for_status()

            # Parse and print the JSON response
            data = response.json()
            print("\n--- Detection Results ---")
            pprint(data)
            print("-------------------------\n")

    except requests.exceptions.RequestException as e:
        print(f"\nAn error occurred while communicating with the API: {e}")
    except FileNotFoundError:
        print(f"\nError: The file '{image_path}' was not found.")
    except Exception as e:
        print(f"\nAn unexpected error occurred: {e}")

if __name__ == "__main__":
    test_detection_api(IMAGE_PATH)
```

To run the test, execute:

Bash

```
pip install requests
python test_client.py
```

A successful run will print a formatted JSON object containing a list of detections, each with a bounding box, confidence score, and class label. This output confirms that every component of the complex system—from the host NVIDIA driver, through the WSL2 GPU passthrough, the Docker container, the FastAPI server, and the TensorRT inference engine—is functioning correctly in unison.

Conclusion

This report has provided a comprehensive, end-to-end blueprint for deploying Baidu's RT-DETR model as a GPU-accelerated, containerized API on a Windows 11 system using WSL2. By architecting the solution in distinct, verifiable stages—from establishing the foundational host environment to optimizing the model with TensorRT, implementing a robust FastAPI service, and encapsulating it within a Docker container—a production-grade inference endpoint has been successfully constructed. The final verification script demonstrates a fully functional system that meets the initial requirement of a network-accessible service for object detection. This methodology not only provides a direct solution but also imparts the underlying principles of modern MLOps, enabling the adaptation and extension of this powerful technology stack to future computer vision challenges.

Works cited

1. CUDA on WSL User Guide — CUDA on WSL 13.0 documentation, accessed August 21, 2025, https://docs.nvidia.com/cuda/wsl-user-guide/index.html
2. Install NVIDIA Container Toolkit on WSL2 - GitHub Gist, accessed August 21, 2025, https://gist.github.com/atinfinity/f9568aa9564371f573138712070f5bad
3. lyuwenyu/RT-DETR: [CVPR 2024] Official RT-DETR (RTDETR paddle pytorch), Real-Time DEtection TRansformer, DETRs Beat YOLOs on Real-time Object Detection. - GitHub, accessed August 21, 2025, https://github.com/lyuwenyu/RT-DETR
4. DETRs Beat YOLOs on Real-time Object Detection, accessed August 21, 2025, https://zhao-yian.github.io/RTDETR/
5. models/rtdetr/ · ultralytics · Discussion #8113 - GitHub, accessed August 21, 2025, https://github.com/orgs/ultralytics/discussions/8113
6. Baidu's RT-DETR: A Vision Transformer-Based Real-Time Object Detector, accessed August 21, 2025, https://docs.ultralytics.com/models/rtdetr/
7. Install Ultralytics, accessed August 21, 2025, https://docs.ultralytics.com/quickstart/
8. nanmi/RT-DETR-Deploy: Deploy RT-EDTR with onnx from paddlepaddle framwork and graph cut - GitHub, accessed August 21, 2025, https://github.com/nanmi/RT-DETR-Deploy
9. sithu31296/PyTorch-ONNX-TRT: TensortRT installation and Conversion from PyTorch Models - GitHub, accessed August 21, 2025, https://github.com/sithu31296/PyTorch-ONNX-TRT
10. RTDETRv2 vs PP-YOLOE+: Detailed Technical Comparison - Ultralytics YOLO Docs, accessed August 21, 2025, https://docs.ultralytics.com/compare/rtdetr-vs-pp-yoloe/
11. Convert a tensor RT engine file back to source Onnx file, or pytorch model wights, accessed August 21, 2025, https://stackoverflow.com/questions/75688465/convert-a-tensor-rt-engine-file-b

ack-to-source-onnx-file-or-pytorch-model-wight
12. Deploying a Vision Transformer Deep Learning Model with FastAPI in Python, accessed August 21, 2025, https://pyimagesearch.com/2024/09/23/deploying-a-vision-transformer-deep-learning-model-with-fastapi-in-python/
13. Boost Your AI Model with FastAPI: A Quick Start Guide, accessed August 21, 2025, https://pub.aimind.so/boost-your-ai-model-with-fastapi-a-quick-start-guide-dccf345698c3
14. PyTorch meets FastAPI and Docker: Streamlining Deep Learning Model Deployment | by Zach Riane Machacon | Medium, accessed August 21, 2025, https://medium.com/@zachriane/pytorch-meets-fastapi-an-streamlining-deep-learning-model-deployment-946b14eb4ddc
15. Build and Deploy Custom Docker Images for Object Recognition - Towards AI, accessed August 21, 2025, https://pub.towardsai.net/build-and-deploy-custom-docker-images-for-object-recognition-d0d127b2603b
16. How to Run YoloV5 Real-Time Object Detection on Pytorch with Docker on NVIDIA® Jetson™ Modules - Forecr.io, accessed August 21, 2025, https://www.forecr.io/blogs/ai-algorithms/how-to-run-yolov5-real-time-object-detection-on-pytorch-with-docker-on-nvidia-jetson-modules
17. PyTorch With Docker. If you are interested in deep learning... | by Zaher Abd Ulmaula | Medium, accessed August 21, 2025, https://medium.com/@zaher88abd/pytorch-with-docker-b791edd67850
18. Docker Image - pytorch, accessed August 21, 2025, https://hub.docker.com/r/pytorch/pytorch

980 docker run --rm -it --gpus 'device=1' -v $(pwd):/workspace nvcr.io/nvidia/pytorch:25.08-py3 bash
19. 981 mkdir app
20. 982 mkdir models
21. 983 vim Dockerfile
22. 984 vim .dockerignore
23. 985 vim requirements.txt
24. 986 mv ../rtdetr-l.engine ./models/
25. 987 cd models/
26. 988 ls
27. 989 cd ../
28. 990 cd app
29. 991 vim inference.py
30. 992 vim main.py
31. 993 docker build -t rtdetr-api .
32. 994 cd ../
33. 995 docker build -t rtdetr-api .
34. 996 vim Dockerfile

35. 997 docker build -t rtdetr-api .
36. 998 docker run --rm -it --gpus 'device=1' -p 8000:8000 rtdetr-api
37. 999 vim app/main.py
38. 1000 vim app/inference.py
39. 1001 docker run --rm -it --gpus 'device=1' -p 8000:8000 rtdetr-api
40. 1002 vim app/inference.py
41. 1003 docker run --rm -it --gpus 'device=1' -p 8000:8000 rtdetr-api
42. 1004 vim app/main.py
43. 1005 tree -d -L 2
44. 1006 tree -L 2
45. 1007 cd app
46. 1008 touch __init__.py
47. 1009 cd ../
48. 1010 docker run --rm -it --gpus 'device=1' -p 8000:8000 rtdetr-api
49. 1011 vim Dockerfile
50. 1012 docker ps -a
51. 1013 docker run --rm -it --gpus 'device=1' -p 8000:8000 rtdetr-api
52. 1014 vim Dockerfile
53. 1015 docker build -t rtdetr-api .
54. 1016 docker run --rm -it --gpus 'device=1' -p 8000:8000 rtdetr-api
55. 1017 vim app/inference.py
56. 1018 rm app/inference.py
57. 1019 vim app/inference.py
58. 1020 docker build -t rtdetr-api .
59. 1021 docker run --rm -it --gpus 'device=1' -p 8000:8000 rtdetr-api
60. 1022 ls
61. 1023 vim Dockerfile
62. 1024 ls
63. 1025 cp rtdetr-l.engine ./models/rtdetr-l.engine
64. 1026 vim Dockerfile
65. 1027 docker build -t rtdetr-api .
66. 1028 docker run --rm -it --gpus 'device=1' -p 8000:8000 rtdetr-api
67. 1029 ls
68. 1030 cd app
69. 1031 ls
70. 1032 mkdir models
71. 1033 cp ../rtdetr-l.engine ./app/
72. 1034 cp ../rtdetr-l.engine ./models/
73. 1035 cd models/
74. 1036 ls
75. 1037 cd ../../
76. 1038 docker build -t rtdetr-api .
77. 1039 docker run --rm -it --gpus 'device=1' -p 8000:8000 rtdetr-api
78. 1040 ls
79. 1041 vim Dockerfile

80. 1042 docker build --no-cache -t rtdetr:ok .
81. 1043 docker run --rm rtdetr:ok bash -lc 'ls -lah /opt/app/models && sha256sum /opt/app/models/rtdetr-l.engine'
82. 1044 docker run --rm -it --gpus 'device=1' -p 8000:8000 rtdetr-api
83. 1045 vim Dockerfile
84. 1046 docker run --rm -it --gpus 'device=1' -p 8000:8000 rtdetr-api
85. 1047 docker build --no-cache -t rtdetr:ok .
86. 1048 vim Dockerfile
87. 1049 docker run --rm rtdetr-api bash -lc 'readlink -f /app; ls -l /app/models/rtdetr-l.engine'
88. 1050 docker build --no-cache -t rtdetr:ok .
89. 1051 docker run --rm rtdetr-api bash -lc 'readlink -f /app; ls -l /app/models/rtdetr-l.engine'
90. 1052 vim Dockerfile
91. 1053 docker build --no-cache -t rtdetr:ok .
92. 1054 docker run --rm rtdetr:ok bash -lc   'echo "-> /app resolves to: $(readlink -f /app)"; \
93.   ls -lah /app/models; \
94.   sha256sum /app/models/rtdetr-l.engine'
95. 1055 docker run --rm -it --gpus 'device=1' -p 8000:8000 rtdetr-api
96. 1056 docker build --no-cache -t rtdetr-api:latest .
97. 1057 docker run --rm -it --gpus 'device=1' -p 8000:8000 rtdetr-api:latest
98. 1058 vim app/inference.py
99. 1059 docker build --no-cache -t rtdetr-api:latest .
100. 1060 docker run --rm -it --gpus 'device=1' -p 8000:8000 rtdetr-api:latest
101. 1061 vim Dockerfile
102. 1062 vim app/inference.py
103. 1063 vim app/main.py
104. 1064 history
105.