Case Study: Vision-Language-Action Models for Autonomous Port Terminal Tractors

Section 1: Industry Context and Business Problem

Industry: Maritime Container Logistics

The global container shipping industry moves USD 14 trillion worth of goods annually across 800+ ports worldwide. A container port is a controlled but intensely dynamic environment: terminal tractors (yard trucks) shuttle 40-foot containers between berths where ships dock, storage yards where containers are stacked, and truck gates where cargo exits the port. A single large port like Long Beach handles 9.5 million twenty-foot equivalent units (TEUs) per year, roughly one container movement every three seconds around the clock.

The operational backbone of this system is the terminal tractor driver. These drivers navigate a maze of container stacks arranged in grid blocks (labeled A through Z, rows 1 through 50), dodge overhead gantry cranes swinging 30-ton loads, yield to maintenance vehicles, and follow real-time dispatcher instructions that change by the minute as ship schedules shift and berth allocations are reassigned.

Company Profile: PacificTow Robotics

Founded: 2022, Long Beach, California **Team:** 87 employees (42 engineers, 12 ML researchers, 8 safety/ops, 25 business/support) **Funding:** Series B, USD 62M raised (Lux Capital lead, Maersk Growth, Toyota Ventures) **Product:** Autonomous terminal tractor retrofit kit — a sensor suite + compute module that converts existing Kalmar or Terberg yard trucks into Level 4 autonomous vehicles within the port perimeter **Current deployment:** Pilot program at a single terminal in the Port of Long Beach, operating 6 autonomous tractors alongside 140 human-driven ones

Business Challenge

PacificTow's current autonomous system uses a traditional modular pipeline: LiDAR-based perception, rule-based path planning on a precomputed port map, and PID-controlled trajectory following. The system works well for **static, pre-planned routes** — moving a container from a known yard position to a known berth along a fixed path.

But the real port is not static. The system breaks down in three critical scenarios:

1. **Dynamic rerouting.** Dispatchers issue real-time instructions via radio: "Container MSCU1234567 was supposed to go to Berth 3, but the vessel is delayed. Divert to Yard Block G, Row 24, Tier 2 instead. Also, avoid Row 12 through 15 — crane maintenance until 1400."

The current system cannot parse these natural language instructions. A human operator must manually reprogram the route, which takes 4-7 minutes per reroute. On a busy day, there are 200+ reroutes across the terminal.

2. **Unstructured navigation.** The port map assumes containers are neatly stacked in their designated positions. In reality, containers are frequently misplaced, temporarily staged in non-standard locations, or partially blocking access lanes. A human driver sees the obstruction and navigates around it. The current system stops and requests human intervention, causing a 12-minute average delay per incident.

3. **Safety around workers.** Port workers walk through container corridors for inspections, lashing, and maintenance. They do not follow predictable paths. The current LiDAR-only detection system has a 94% worker detection rate at close range but struggles with occlusion (a worker stepping out from behind a container stack) and adverse conditions (rain, fog, low-angle sun creating long shadows). The 6% miss rate is unacceptable for a system moving 30-ton loads.

Financial Impact

- **Labor cost:** Terminal tractor drivers earn USD 85,000-110,000/year with benefits. The pilot terminal employs 140 drivers across three shifts. Full automation could reduce this to 20 remote supervisors, saving USD 12.6M/year at this terminal alone.
- **Throughput loss from rerouting delays:** At 200 reroutes/day averaging 4 minutes of downtime each, the terminal loses 13.3 tractor-hours daily. At the terminal's throughput rate of USD 47/container-move, this costs USD 1.8M/year in lost capacity.
- **Incident costs:** Each human intervention event (stopping, waiting for remote operator, resuming) costs an average of USD 180 in lost throughput. At 35 events per tractor per shift, the 6 autonomous tractors generate USD 68,000/month in intervention costs alone.
- **Insurance and safety:** One serious injury incident costs USD 2-5M in direct liability plus regulatory review delays. The industry target is zero incidents.

Constraints

- **Compute budget:** Each tractor carries an NVIDIA Orin module (275 TOPS INT8). Any VLA model must run inference within this envelope. Cloud offloading is not an option — port WiFi is unreliable near metal container stacks.
- **Latency:** Tractors operate at up to 25 km/h in the port. At this speed, the vehicle travels 7 meters per second. The system must produce a new trajectory within 200ms to maintain safe operation. The language reasoning component can run at a lower cadence (1-2 Hz) in a dual-system architecture.
- **Data availability:** PacificTow has collected 4,200 hours of driving data from their 6 pilot tractors over 14 months, including multi-camera video (4 cameras per tractor at 10 Hz), GPS trajectories, and dispatcher radio transcripts (partially transcribed). They also have access to 800 hours of human-driven tractor data from their partner terminal.

- **Privacy/compliance:** Port operations fall under Maritime Transportation Security Act (MTSA) and USCG regulations. Camera footage that captures ship manifests, container IDs, or worker faces must be handled per MTSA data handling requirements. No port operational data may leave US-based servers.
- **Deployment environment:** Outdoor port environment with rain, fog, dust, salt air corrosion, temperature range of 5-40 C, and 24/7 operation including nighttime with floodlighting. Vibration from rough port surfaces. No controlled indoor environment.
- **Team expertise:** 12 ML researchers with experience in computer vision and NLP, but no prior experience with diffusion models or VLA architectures. 3 robotics engineers with ROS2 experience for vehicle integration.

Section 2: Technical Problem Formulation

Problem Type: Conditional Trajectory Generation

The core task is to generate a safe, smooth trajectory for the terminal tractor given: - What the tractor currently sees (multi-camera images) - What the dispatcher is telling it to do (natural language instruction) - Where the tractor has been recently (egomotion history)

This is fundamentally a **conditional generation problem**, not a classification or regression problem. Here is why.

Why not classification? One might try to discretize the action space into a fixed set of maneuvers (go straight, turn left, turn right, stop). But port navigation requires continuous, precise trajectories through narrow corridors. The space between two container stacks is often only 0.5 meters wider than the tractor itself. Discrete action categories cannot capture this precision.

Why not regression? A regression model trained with mean squared error would predict the average trajectory. But in many port scenarios, there are multiple valid paths. At a corridor intersection, the tractor could turn left or right — both valid depending on which is faster. A regression model would average these and drive straight into a container stack. This is the **multimodality problem** that the article discusses in the context of diffusion decoders.

Why conditional generation? We need a model that: 1. Can produce **multiple valid trajectories** (multimodality) 2. Can condition on **both visual context and language instructions** (multi-modal input) 3. Generates **smooth, continuous** waypoint sequences (not discrete action tokens)

This maps directly to the VLA architecture described in the article: a vision encoder processes camera images, a language model reasons about the scene and instructions, and a diffusion-based action decoder generates trajectory waypoints.

Input Specification

The model receives three input modalities:

1. Multi-camera images — 4 cameras mounted on the tractor (front, rear, left, right), each capturing 640x480 RGB images at 10 Hz. We use a 0.4-second history window (4 frames per camera), giving a tensor of shape $(4 \text{ cameras} \times 4 \text{ frames} \times 3 \times 480 \times 640)$. The temporal history allows the model to perceive motion — a worker walking, a crane swinging, another tractor approaching.

2. Dispatcher instruction — A natural language string, typically 10-50 tokens. Examples: - "Pick up container at Block D, Row 18. Deliver to Berth 2, Crane 4." - "Reroute: avoid Rows 12 through 15 in Block C, crane maintenance." - "Priority load. Proceed to Berth 1 via the express lane."

The language input carries information that is **not available in the visual input** — the destination, priority level, and restricted zones are not visible from the tractor's cameras.

3. Egomotion history — The tractor's own recent trajectory: 16 waypoints over the past 1.6 seconds, each consisting of (x, y, θ) (position and heading). Shape: $(16, 3)$. This tells the model how the tractor is currently moving — its speed, heading, and recent steering. Without this, the model would not know whether the tractor is currently turning or going straight, which is critical for generating a smooth continuation.

Output Specification

The model produces two outputs:

1. Future trajectory — 32 waypoints over the next 3.2 seconds at 10 Hz. Each waypoint is (x, y, θ) in the tractor's local coordinate frame. Shape: $(32, 3)$. We chose 3.2 seconds (shorter than Alpamayo's 6.4 seconds) because port speeds are lower (25 km/h max vs. highway speeds) and the environment changes more rapidly (tight turns, frequent stops).

2. Reasoning trace — A natural language string explaining the model's decision: "Proceeding through Block D corridor. Worker detected at Row 16 — slowing and maintaining right clearance. Destination Row 18 is two rows ahead." This trace is generated by the language model component and is logged for safety auditing.

The trajectory representation uses the tractor's **local coordinate frame** (origin at current position, x-axis along current heading). This is important because it makes the output independent of the tractor's global position, allowing the model to generalize across different areas of the port.

Mathematical Foundation

The VLA architecture combines three mathematical frameworks. Let us build each from first principles.

Framework 1: Vision Encoding via Patch Embedding and Self-Attention

The vision encoder converts a raw image $I \in \mathbb{R}^{3\times H\times W}$ into a sequence of feature tokens. The image is divided into $N = \frac{H\times W}{P^2}$ non-overlapping patches of size $P \times P$. Each patch $p_i \in \mathbb{R}^{3P^2}$ is linearly projected into a d-dimensional embedding:

$$z_i = W_E \cdot \text{flatten}(p_i) + b_E, \quad W_E \in \mathbb{R}^{d\times 3P^2}$$

Learnable positional embeddings $e_i \in \mathbb{R}^d$ are added so the model knows where each patch came from spatially: $z_i \leftarrow z_i + e_i$.

The sequence of patch embeddings is then processed by L transformer layers with multi-head self-attention:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

where $Q = ZW_Q$, $K = ZW_K$, $V = ZW_V$ are learned linear projections. The scaling by $\sqrt{d_k}$ prevents the dot products from growing large and pushing the softmax into regions with tiny gradients — a detail that matters enormously for stable training.

Why self-attention for vision? The critical property is that self-attention allows every patch to attend to every other patch. When the front camera sees a worker near Row 16, the attention mechanism allows the model to simultaneously consider the container stacks on both sides (from left and right cameras) to determine available clearance. A convolutional network would require many layers to build this global context; a transformer achieves it in a single attention layer.

Framework 2: Behavioral Cloning as Maximum Likelihood

Training the VLA on expert demonstrations is formalized as maximum likelihood estimation. Given a dataset of N expert demonstrations $\mathcal{D} = \{(o_i, \ell_i, \tau_i^*)\}_{i=1}^N$ where o_i is the observation (images + egomotion), ℓ_i is the language instruction, and τ_i^* is the expert trajectory, we maximize:

$$\theta^* = \arg\max_\theta \sum_{i=1}^N \log p_\theta(\tau_i^* \mid o_i, \ell_i)$$

This is equivalent to minimizing the negative log-likelihood, which for Gaussian-distributed trajectory predictions reduces to:

$$\mathcal{L}_{\text{BC}} = \frac{1}{N}\sum_{i=1}^N \left\|\tau_i^* - \hat{\tau}_\theta(o_i, \ell_i)\right\|_2^2$$

The mean squared error emerges naturally from the assumption that the expert's trajectory is the mean of a Gaussian, and we are finding the parameters that make this Gaussian assign maximum probability to the observed data.

The problem with pure MSE behavioral cloning is precisely the multimodality issue. If the expert data contains two trajectories for the same observation — one turning left, one turning right — the MSE-optimal prediction is the average, which goes straight into a wall. This motivates the diffusion-based approach.

Framework 3: Diffusion-Based Trajectory Generation

The diffusion action decoder solves the multimodality problem. Instead of predicting a single trajectory, it learns the full **distribution** of valid trajectories and samples from it.

Forward process (adding noise). Given a clean expert trajectory τ_0, we progressively add Gaussian noise over T steps:

$$\tau_t = \sqrt{\bar{\alpha}_t} \cdot \tau_0 + \sqrt{1 - \bar{\alpha}_t} \cdot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

where $\bar{\alpha}_t = \prod_{s=1}^{t}(1 - \beta_s)$ is the cumulative noise schedule. At $t = 0$, $\bar{\alpha}_0 \approx 1$ and we have the original trajectory. At $t = T$, $\bar{\alpha}_T \approx 0$ and we have pure noise. The trajectory has been completely destroyed.

Reverse process (denoising). A neural network ϵ_ϕ learns to predict the noise at each step, conditioned on the current noisy trajectory, the timestep, and the VLM's output features c (which encode both visual context and language understanding):

$$\epsilon_\phi(\tau_t, t, c) \approx \epsilon$$

The training loss is:

$$\mathcal{L}_{\text{diff}} = \mathbb{E}_{t, \tau_0, \epsilon}\left[\left\|\epsilon - \epsilon_\phi(\sqrt{\bar{\alpha}_t}\tau_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon,\ t,\ c)\right\|_2^2\right]$$

Let us unpack this loss term by term:

- ϵ is the actual noise that was added. This is known during training because we added it ourselves.
- $\epsilon_\phi(\cdot)$ is the model's prediction of that noise. The model sees only the noisy trajectory and must guess what noise was added.
- The expectation is over random timesteps t (uniform from 1 to T), random clean trajectories τ_0 (from the dataset), and random noise ϵ (standard Gaussian).
- The MSE between predicted and actual noise drives the model to learn accurate denoising at every noise level.

Why does this solve multimodality? At sampling time, we start from pure noise $\tau_T \sim \mathcal{N}(0, I)$ and iteratively denoise. Different random starting points lead to different valid trajectories. The diffusion model has learned the full distribution, not just the mean. For the corridor intersection scenario, some noise samples will be denoised into left turns, others into right turns — both valid. None will produce the dangerous average.

Connecting the math to the business problem. The $\sqrt{\bar{\alpha}_t}$ coefficient controls how much original trajectory signal is retained at step t. Early in denoising (large t), the model makes coarse decisions — "am I going left or right?" Later (small t), it refines fine details — "exactly how tight should this turn be?" This coarse-to-fine generation is what produces the smooth, precise trajectories needed for navigating 0.5-meter clearances between container stacks.

Loss Function

The complete training loss combines three terms:

$$\mathcal{L} = \lambda_{\text{diff}}\mathcal{L}_{\text{diff}} + \lambda_{\text{lang}}\mathcal{L}_{\text{lang}} + \lambda_{\text{col}}\mathcal{L}_{\text{col}}$$

Term 1: Diffusion denoising loss $\mathcal{L}_{\text{diff}}$ (weight $\lambda_{\text{diff}} = 1.0$)

This is the core trajectory generation loss described above. It trains the diffusion decoder to produce accurate trajectories. If you remove this term, the model cannot generate trajectories at all.

Term 2: Language modeling loss $\mathcal{L}_{\text{lang}}$ (weight $\lambda_{\text{lang}} = 0.1$)

$$\mathcal{L}_{\text{lang}} = -\sum_{j=1}^{|\text{reasoning}|} \log p_\theta(w_j \mid w_{<j}, o, \ell)$$

This is the standard autoregressive cross-entropy loss on the reasoning trace. It trains the language model to produce coherent reasoning about the driving scene. If you remove this term, the model still generates trajectories but loses interpretability — you cannot audit why the tractor made a specific decision, which is unacceptable for safety certification.

The weight of 0.1 (rather than 1.0) reflects that trajectory accuracy is more important than reasoning fluency. Increasing this weight improves reasoning quality but slightly degrades trajectory precision because the language model's gradients compete with the diffusion decoder's gradients for the shared VLM backbone.

Term 3: Collision penalty $\mathcal{L}_{\text{col}}$ (weight $\lambda_{\text{col}} = 5.0$)

$$\mathcal{L}_{\text{col}} = \frac{1}{K}\sum_{k=1}^{K} \max\left(0,\ d_{\text{safe}} - d(\hat{\tau}_k, \mathcal{O})\right)^2$$

where $d(\hat{\tau}_k, \mathcal{O})$ is the minimum distance between the k-th predicted waypoint and the nearest obstacle $\mathcal{O}$, and $d_{\text{safe}} = 0.5\text{m}$ is the minimum safe clearance. This is a hinge loss — it contributes zero when the trajectory is safely clear of obstacles, and penalizes quadratically when the trajectory violates the safety margin.

If you remove this term, the model still learns to avoid collisions implicitly from the expert data (experts do not crash), but it lacks an explicit safety margin. The high weight of 5.0 reflects PacificTow's zero-incident safety requirement. Increasing this weight further would make the tractor overly conservative — maintaining excessive clearance and reducing throughput.

Evaluation Metrics

Primary metrics:

1. **Average Displacement Error (ADE):** Mean L2 distance between predicted and ground-truth waypoints, averaged over all timesteps. Target: < 0.3 meters. This measures overall trajectory accuracy. A human driver's average displacement from the "optimal" path is approximately 0.2-0.4 meters in port environments.

2. **Final Displacement Error (FDE):** L2 distance at the final predicted waypoint (t = 3.2s). Target: < 0.8 meters. This measures whether the trajectory ends in approximately the right place — critical for container pickup alignment.

3. **Collision rate:** Percentage of predicted trajectories that come within 0.3 meters of any obstacle. Target: 0.0%. Non-negotiable for safety.

Secondary metrics:

1. **Instruction following accuracy:** Does the model navigate to the correct destination as specified in the dispatcher instruction? Measured as the percentage of trajectories that terminate within 2 meters of the intended block/row. Target: > 95%.

2. **Smoothness (jerk):** Average jerk (third derivative of position) along the trajectory. Target: < 2.0 m/s^3. Excessive jerk means uncomfortable or unsafe motion — containers can shift on the chassis if the tractor moves too abruptly.

3. **Inference latency:** Wall-clock time from input to trajectory output. Target: < 200ms on NVIDIA Orin. Measured end-to-end including vision encoding, language processing, and diffusion sampling.

Baseline

Naive baseline: GPS waypoint following with PID control. PacificTow's current system precomputes optimal routes on a static port map and follows them with a PID controller. The dispatcher must manually enter route changes through a tablet interface (4-7 minutes per reroute).

Performance: - ADE: 0.15m (excellent on known routes, because paths are pre-optimized) - FDE: 0.20m - Collision rate: 0.8% (collisions with temporarily misplaced containers not on the map) - Instruction following: 0% (cannot process natural language at all) - Reroute latency: 4-7 minutes (human operator in the loop)

The baseline excels at following known routes but completely fails at dynamic adaptation. It cannot understand dispatcher instructions, cannot handle map discrepancies, and requires human intervention for any deviation from the precomputed plan. The 0.8% collision rate with unmapped obstacles is the most critical failure — this translates to approximately 12 near-miss incidents per day across the fleet.

Why This Concept: VLA as the Technical Solution

The VLA architecture is the right approach for three fundamental reasons:

1. **Multi-modal fusion is inherent to the problem.** The tractor must simultaneously process visual information (what it sees), linguistic information (what the dispatcher says), and kinematic information (how it is moving). A VLA unifies these modalities in a single model, allowing cross-modal reasoning: "The dispatcher says avoid Row 12, and I can see Row 12 markers in my left camera, so I should plan a trajectory that stays to the right."

2. **The language backbone provides world knowledge.** Port operations involve domain-specific reasoning: "crane maintenance means the area directly below the crane is restricted, plus a 5-meter safety buffer on each side." A pre-trained language model has encountered these concepts in its training data and can reason about them without explicit programming.

3. **Diffusion-based action generation handles multimodality.** At corridor intersections, there are often multiple valid routes to the same destination. The diffusion decoder can represent this distribution of valid trajectories and sample from it, rather than averaging to a dangerous compromise trajectory.

Technical Constraints

- **Model size:** Must fit within NVIDIA Orin's 32GB memory. Target total model size: < 3B parameters (significantly smaller than Alpamayo's 10.5B, reflecting the constrained compute environment).
- **Inference latency:** < 200ms end-to-end on Orin. The diffusion decoder must use DDIM sampling with 8-10 steps (not the full 100 steps used in training).
- **Training compute:** 8x NVIDIA A100 GPUs for 72 hours (PacificTow's cloud budget). This constrains the model to ~1B trainable parameters with the VLM backbone frozen.
- **Data volume:** 5,000 hours of driving data (4,200 from PacificTow + 800 from partner terminal). Approximately 180 million image frames and 50,000 transcribed dispatcher instructions.

Section 3: Implementation Notebook Structure

This notebook builds a simplified VLA for port terminal tractor navigation. The student implements the core components — vision encoding, language conditioning, and diffusion-based trajectory generation — on a synthetic port environment dataset that captures the essential structure of the real problem.

3.1 Data Acquisition Strategy

Dataset: Synthetic Port Environment

We generate a synthetic dataset that models the core challenges of port navigation: grid-structured corridors, static obstacles (container stacks), dynamic obstacles (workers, other tractors), and paired natural language instructions.

The synthetic environment is a 200m x 200m port yard with: - 10 container blocks (A through J), each containing 20 rows of containers - 3-meter-wide corridors between rows - Random worker positions (1-5 workers per scene) - Random temporary obstacles (misplaced containers, equipment)

Each data sample consists of: - A simplified "birds-eye-view" image (top-down occupancy grid, 128x128 pixels) representing what the tractor's cameras see after processing - A text instruction (e.g., "Navigate to Block D, Row 14") - The expert trajectory (32 waypoints, generated by A* pathfinding with smoothing) - Obstacle positions for collision checking

This synthetic approach is appropriate because: 1. Students can generate unlimited training data on a Colab T4 2. The core VLA learning problem (fusing vision + language + action) is preserved 3. The grid structure of a real port is faithfully represented 4. We avoid privacy issues with real port footage

Setup and data generation code:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt
from torch.utils.data import Dataset, DataLoader
from typing import Tuple, List, Dict
import random
import math

# Reproducibility
SEED = 42
torch.manual_seed(SEED)
np.random.seed(SEED)
random.seed(SEED)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")
```

```
# --- Port Environment Configuration ---
PORT_SIZE = 128          # Grid size (128x128 = ~200m x 200m at 1.5m/cell)
NUM_BLOCKS = 8           # Container blocks (A-H)
ROWS_PER_BLOCK = 12      # Rows per block
CORRIDOR_WIDTH = 2       # Grid cells between rows (~3 meters)
BLOCK_GAP = 6            # Grid cells between blocks (~9 meters)
NUM_WAYPOINTS = 32       # Trajectory length (3.2 seconds at 10Hz)
MAX_WORKERS = 4          # Max dynamic obstacles per scene
NUM_INSTRUCTION_TYPES = 4  # Types of dispatcher instructions


class PortEnvironment:
    """Generates synthetic port yard layouts with container blocks, corridors, and obstacles."""

    BLOCK_NAMES = [chr(65 + i) for i in range(NUM_BLOCKS)]  # A, B, C, ...

    def __init__(self, size: int = PORT_SIZE):
        self.size = size
        self.grid = np.zeros((size, size), dtype=np.float32)
        self.block_positions = {}  # block_name -> (row_start, col_start, row_end, col_end)
        self._build_layout()

    def _build_layout(self):
```

```
        """Place container blocks in a grid pattern."""
        block_width = 8   # cells wide per block
        block_height = ROWS_PER_BLOCK * (1 + CORRIDOR_WIDTH)

        start_x = 10
        start_y = 10
        col_idx = 0
        row_idx = 0

        for i, name in enumerate(self.BLOCK_NAMES):
            bx = start_x + col_idx * (block_width + BLOCK_GAP)
            by = start_y + row_idx * (block_height + BLOCK_GAP)

            if by + block_height >= self.size - 10:
                col_idx += 1
                row_idx = 0
                bx = start_x + col_idx * (block_width + BLOCK_GAP)
                by = start_y

            self.block_positions[name] = (by, bx, by + block_height, bx + block_width)

            # Fill container rows (leave corridors)
            for r in range(ROWS_PER_BLOCK):
                ry = by + r * (1 + CORRIDOR_WIDTH)
                self.grid[ry:ry+1, bx:bx+block_width] = 1.0  # container row

            row_idx += 1

    def get_destination(self, block_name: str, row: int) -> Tuple[int, int]:
        """Get the grid coordinates for a block/row destination."""
        if block_name not in self.block_positions:
            block_name = random.choice(list(self.block_positions.keys()))
        by, bx, _, bx_end = self.block_positions[block_name]
        row = min(row, ROWS_PER_BLOCK - 1)
        dest_y = by + row * (1 + CORRIDOR_WIDTH) + 1  # in the corridor
        dest_x = bx + (bx_end - bx) // 2  # center of block
        return (dest_y, dest_x)

    def add_random_workers(self, n_workers: int) -> List[Tuple[int, int]]:
        """Add random worker positions in corridors."""
        workers = []
        attempts = 0
        while len(workers) < n_workers and attempts < 100:
            y = random.randint(5, self.size - 5)
            x = random.randint(5, self.size - 5)
            if self.grid[y, x] == 0:  # only in free space
                workers.append((y, x))
            attempts += 1
        return workers

    def render(self, tractor_pos=None, trajectory=None, workers=None, destination=None):
        """Render the port as a 3-channel image (128x128x3)."""
        img = np.zeros((self.size, self.size, 3), dtype=np.float32)
        # Channel 0: container blocks (static obstacles)
        img[:, :, 0] = self.grid
        # Channel 1: dynamic obstacles (workers)
        if workers:
            for wy, wx in workers:
                y_lo, y_hi = max(0, wy-1), min(self.size, wy+2)
                x_lo, x_hi = max(0, wx-1), min(self.size, wx+2)
                img[y_lo:y_hi, x_lo:x_hi, 1] = 1.0
        # Channel 2: destination marker
        if destination:
            dy, dx = destination
            y_lo, y_hi = max(0, dy-2), min(self.size, dy+3)
            x_lo, x_hi = max(0, dx-2), min(self.size, dx+3)
            img[y_lo:y_hi, x_lo:x_hi, 2] = 1.0
        return img
```

```
# --- Trajectory Generation (Expert Planner) ---

def bresenham_line(y0, x0, y1, x1):
    """Generate grid cells along a line using Bresenham's algorithm."""
    points = []
    dx = abs(x1 - x0)
```

```
    dy = abs(y1 - y0)
    sx = 1 if x0 < x1 else -1
    sy = 1 if y0 < y1 else -1
    err = dx - dy
    while True:
        points.append((y0, x0))
        if y0 == y1 and x0 == x1:
            break
        e2 = 2 * err
        if e2 > -dy:
            err -= dy
            x0 += sx
        if e2 < dx:
            err += dx
            y0 += sy
    return points


def plan_trajectory(grid, start, end, num_waypoints=NUM_WAYPOINTS):
    """
    Plan a trajectory from start to end, avoiding obstacles.
    Uses a simplified approach: straight-line with corridor-aware waypoint adjustment.
    Returns normalized waypoints in [-1, 1] relative to start position.
    """
    sy, sx = start
    ey, ex = end

    # Generate raw waypoints via linear interpolation
    raw_y = np.linspace(sy, ey, num_waypoints + 10)
    raw_x = np.linspace(sx, ex, num_waypoints + 10)

    # Add slight noise and smooth for realistic trajectories
    noise_scale = 0.3
    raw_y += np.random.randn(len(raw_y)) * noise_scale
    raw_x += np.random.randn(len(raw_x)) * noise_scale

    # Simple smoothing (moving average)
    kernel = np.ones(5) / 5
    raw_y = np.convolve(raw_y, kernel, mode='same')
    raw_x = np.convolve(raw_x, kernel, mode='same')

    # Subsample to desired number of waypoints
    indices = np.linspace(0, len(raw_y) - 1, num_waypoints).astype(int)
    traj_y = raw_y[indices]
    traj_x = raw_x[indices]

    # Normalize: convert to offsets from start, scale to [-1, 1] range
    traj_y = (traj_y - sy) / (PORT_SIZE / 2)
    traj_x = (traj_x - sx) / (PORT_SIZE / 2)

    # Compute heading at each waypoint
    dy = np.gradient(traj_y)
    dx = np.gradient(traj_x)
    theta = np.arctan2(dy, dx)

    trajectory = np.stack([traj_x, traj_y, theta], axis=-1).astype(np.float32)
    return trajectory  # shape: (NUM_WAYPOINTS, 3)
```

```
# --- Instruction Generation ---

INSTRUCTION_TEMPLATES = [
    "Navigate to Block {block}, Row {row}.",
    "Pick up container at Block {block}, Row {row}. Proceed via main corridor.",
    "Deliver to Block {block}, Row {row}. Avoid Row {avoid_row} in Block {avoid_block} — maintenance in progress.",
    "Priority load. Go to Block {block}, Row {row} immediately.",
]

def generate_instruction(dest_block: str, dest_row: int, env: PortEnvironment) -> str:
    """Generate a natural language dispatcher instruction."""
    template_idx = random.randint(0, len(INSTRUCTION_TEMPLATES) - 1)
    template = INSTRUCTION_TEMPLATES[template_idx]

    avoid_block = random.choice(env.BLOCK_NAMES)
    avoid_row = random.randint(1, ROWS_PER_BLOCK)
```

```
    instruction = template.format(
        block=dest_block,
        row=dest_row,
        avoid_block=avoid_block,
        avoid_row=avoid_row,
    )
    return instruction
```

```
# --- Dataset Class ---

class PortVLADataset(Dataset):
    """
    Synthetic dataset for port VLA training.
    Each sample: (image, instruction_tokens, egomotion, trajectory, obstacles)
    """

    def __init__(self, num_samples: int = 5000, max_instruction_len: int = 32):
        self.num_samples = num_samples
        self.max_instruction_len = max_instruction_len
        self.env = PortEnvironment()
        self.data = []

        # Build a simple word-to-index vocabulary from templates
        self.vocab = {"<pad>": 0, "<unk>": 1}
        all_words = set()
        for t in INSTRUCTION_TEMPLATES:
            for w in t.replace(".", "").replace(",", "").replace("{", "").replace("}", "").split():
                all_words.add(w.lower())
        for name in self.env.BLOCK_NAMES:
            all_words.add(name.lower())
        for i in range(1, ROWS_PER_BLOCK + 1):
            all_words.add(str(i))
        for i, w in enumerate(sorted(all_words), start=2):
            self.vocab[w] = i
        # Add special tokens for block names and numbers
        for c in "abcdefghijklmnopqrstuvwxyz":
            if c not in self.vocab:
                self.vocab[c] = len(self.vocab)

        self.vocab_size = len(self.vocab)
        self._generate_all()

    def tokenize(self, text: str) -> torch.Tensor:
        """Simple whitespace tokenizer."""
        words = text.replace(".", "").replace(",", "").replace("—", " ").lower().split()
        tokens = [self.vocab.get(w, 1) for w in words]  # 1 = <unk>
        # Pad or truncate
        if len(tokens) < self.max_instruction_len:
            tokens += [0] * (self.max_instruction_len - len(tokens))
        else:
            tokens = tokens[:self.max_instruction_len]
        return torch.tensor(tokens, dtype=torch.long)

    def _generate_all(self):
        """Pre-generate all samples."""
        for _ in range(self.num_samples):
            # Random start position (in a corridor)
            start_y = random.randint(15, PORT_SIZE - 15)
            start_x = random.randint(15, PORT_SIZE - 15)
            while self.env.grid[start_y, start_x] > 0:
                start_y = random.randint(15, PORT_SIZE - 15)
                start_x = random.randint(15, PORT_SIZE - 15)

            # Random destination
            dest_block = random.choice(self.env.BLOCK_NAMES)
            dest_row = random.randint(1, ROWS_PER_BLOCK)
            dest_pos = self.env.get_destination(dest_block, dest_row - 1)

            # Generate components
            workers = self.env.add_random_workers(random.randint(0, MAX_WORKERS))
            image = self.env.render(
                tractor_pos=(start_y, start_x),
                workers=workers,
                destination=dest_pos
            )
```

```
            instruction = generate_instruction(dest_block, dest_row, self.env)
            tokens = self.tokenize(instruction)
            trajectory = plan_trajectory(self.env.grid, (start_y, start_x), dest_pos)

            # Egomotion: simple straight-line history (16 waypoints, 3D)
            ego_history = np.zeros((16, 3), dtype=np.float32)
            ego_history[:, 0] = np.linspace(-0.1, 0, 16)  # slight forward motion

            self.data.append({
                "image": torch.tensor(image).permute(2, 0, 1),  # (3, 128, 128)
                "instruction": tokens,
                "egomotion": torch.tensor(ego_history),
                "trajectory": torch.tensor(trajectory),
                "start": (start_y, start_x),
                "dest": dest_pos,
                "workers": workers,
            })

    def __len__(self):
        return self.num_samples

    def __getitem__(self, idx):
        d = self.data[idx]
        return d["image"], d["instruction"], d["egomotion"], d["trajectory"]


# Generate datasets
print("Generating training dataset (5000 samples)...")
train_dataset = PortVLADataset(num_samples=5000)
print("Generating validation dataset (500 samples)...")
val_dataset = PortVLADataset(num_samples=500)

print(f"Vocabulary size: {train_dataset.vocab_size}")
print(f"Sample image shape: {train_dataset[0][0].shape}")
print(f"Sample instruction shape: {train_dataset[0][1].shape}")
print(f"Sample trajectory shape: {train_dataset[0][3].shape}")
```

TODO: Data Augmentation

Students implement two data augmentation strategies.

```
# TODO 1: Implement Data Augmentation for Port VLA Training

def augment_trajectory_noise(trajectory: torch.Tensor, noise_std: float = 0.01) -> torch.Tensor:
    """
    Add small Gaussian noise to expert trajectories to improve robustness.

    This simulates the natural variation in human driving — two expert drivers
    navigating the same route will produce slightly different trajectories.
    Augmenting with noise prevents the model from overfitting to the exact
    waypoint positions in the training data.

    Args:
        trajectory: Expert trajectory, shape (32, 3) — [x, y, theta] per waypoint
        noise_std: Standard deviation of Gaussian noise to add

    Returns:
        Augmented trajectory with same shape

    Hints:
        Step 1: Generate Gaussian noise with torch.randn_like, scaled by noise_std
        Step 2: Add noise only to the x and y channels (indices 0 and 1), NOT theta
                (heading noise should be recomputed from the noisy x,y positions)
        Step 3: Recompute theta from the noisy x,y using torch.atan2 on the
                finite differences (gradient) of y and x
        Step 4: Clamp x and y to [-1, 1] to keep waypoints in valid range
    """
    # YOUR CODE HERE (approximately 6-8 lines)
    pass


def augment_mirror(image: torch.Tensor, instruction: torch.Tensor,
                   trajectory: torch.Tensor) -> Tuple[torch.Tensor, torch.Tensor, torch.Tensor]:
```

```
    """
    Mirror the scene horizontally — flip the image, negate the x-coordinates
    of the trajectory, and negate the heading angles.

    This effectively doubles the dataset by creating left-right mirror images.
    In a symmetric port layout, a trajectory going left is equally valid as
    one going right.

    Args:
        image: Port scene image, shape (3, 128, 128)
        instruction: Tokenized instruction, shape (32,) — left unchanged
        trajectory: Expert trajectory, shape (32, 3)

    Returns:
        Tuple of (flipped_image, instruction, flipped_trajectory)

    Hints:
        Step 1: Flip the image horizontally using torch.flip on the last dimension
        Step 2: Negate the x-coordinates (index 0) of the trajectory
        Step 3: Negate the theta values (index 2) of the trajectory
        Step 4: Return the instruction unchanged (text is not mirrored)
    """
    # YOUR CODE HERE (approximately 4-5 lines)
    pass
```

```
# --- Verification for TODO 1 ---
# Test augment_trajectory_noise
test_traj = torch.randn(32, 3)
test_traj[:, :2] = test_traj[:, :2].clamp(-0.9, 0.9)

if augment_trajectory_noise is not None and augment_trajectory_noise(test_traj) is not None:
    aug_traj = augment_trajectory_noise(test_traj, noise_std=0.01)
    assert aug_traj.shape == (32, 3), f"Expected shape (32, 3), got {aug_traj.shape}"
    assert not torch.allclose(aug_traj[:, :2], test_traj[:, :2]), "Noise was not added to x,y"
    assert (aug_traj[:, :2].abs() <= 1.0).all(), "x,y values out of [-1, 1] range"
    print("[PASS] augment_trajectory_noise: shape, noise, and clamping correct")
else:
    print("[SKIP] augment_trajectory_noise not yet implemented")

# Test augment_mirror
test_img = torch.randn(3, 128, 128)
test_inst = torch.randint(0, 50, (32,))
test_traj2 = torch.randn(32, 3)

if augment_mirror is not None and augment_mirror(test_img, test_inst, test_traj2) is not None:
    f_img, f_inst, f_traj = augment_mirror(test_img, test_inst, test_traj2)
    assert f_img.shape == test_img.shape, "Image shape changed"
    assert torch.allclose(f_traj[:, 0], -test_traj2[:, 0]), "x not negated"
    assert torch.allclose(f_traj[:, 2], -test_traj2[:, 2]), "theta not negated"
    assert torch.allclose(f_inst, test_inst), "Instruction should be unchanged"
    print("[PASS] augment_mirror: flip and negation correct")
else:
    print("[SKIP] augment_mirror not yet implemented")
```

3.2 Exploratory Data Analysis

```
# --- Visualize the Port Environment ---

env = train_dataset.env
fig, axes = plt.subplots(1, 3, figsize=(18, 6))

# 1. Port layout
axes[0].imshow(env.grid, cmap='Oranges', origin='lower')
for name, (by, bx, ey, ex) in env.block_positions.items():
    axes[0].text(bx + (ex - bx) / 2, by - 2, f"Block {name}",
                 ha='center', fontsize=8, fontweight='bold')
axes[0].set_title("Port Yard Layout (Container Blocks)")
axes[0].set_xlabel("X (grid cells)")
axes[0].set_ylabel("Y (grid cells)")

# 2. Sample scene with trajectory
sample = train_dataset.data[0]
```

```
img = sample["image"].permute(1, 2, 0).numpy()
axes[1].imshow(img, origin='lower')
traj = sample["trajectory"].numpy()
sy, sx = sample["start"]
# Convert normalized trajectory back to grid coords for plotting
plot_x = traj[:, 0] * (PORT_SIZE / 2) + sx
plot_y = traj[:, 1] * (PORT_SIZE / 2) + sy
axes[1].plot(plot_x, plot_y, 'g-', linewidth=2, label='Expert trajectory')
axes[1].plot(plot_x[0], plot_y[0], 'go', markersize=8, label='Start')
axes[1].plot(plot_x[-1], plot_y[-1], 'r*', markersize=12, label='End')
axes[1].legend(fontsize=8)
axes[1].set_title("Sample Scene + Trajectory")

# 3. Trajectory length distribution
lengths = []
for d in train_dataset.data[:500]:
    t = d["trajectory"].numpy()
    dx = np.diff(t[:, 0])
    dy = np.diff(t[:, 1])
    length = np.sum(np.sqrt(dx**2 + dy**2))
    lengths.append(length)
axes[2].hist(lengths, bins=30, edgecolor='black', alpha=0.7)
axes[2].set_xlabel("Trajectory Length (normalized units)")
axes[2].set_ylabel("Count")
axes[2].set_title("Distribution of Trajectory Lengths")
axes[2].axvline(np.mean(lengths), color='red', linestyle='--', label=f'Mean: {np.mean(lengths):.2f}')
axes[2].legend()

plt.tight_layout()
plt.show()
```

```
# --- Analyze Instruction Distribution ---

from collections import Counter

# Count instruction types
type_counts = Counter()
word_counts = Counter()
for d in train_dataset.data:
    inst = d.get("instruction", None)
    # Reconstruct instruction text from dataset for analysis
    pass  # We analyze templates instead

template_counts = Counter()
for i in range(len(train_dataset)):
    # Approximate: check which template by looking for key phrases
    tokens = train_dataset.data[i]["instruction"]
    template_counts["total"] += 1

print(f"Total training samples: {len(train_dataset)}")
print(f"Total validation samples: {len(val_dataset)}")
print(f"Trajectory shape: {train_dataset[0][3].shape}")
print(f"Trajectory value range: [{train_dataset[0][3].min():.3f}, {train_dataset[0][3].max():.3f}]")
print(f"Mean trajectory length: {np.mean(lengths):.3f}")
print(f"Std trajectory length: {np.std(lengths):.3f}")
```

TODO: EDA Deep Dive

```
# TODO 2: Exploratory Data Analysis

# Analyze and plot the following (write code for each):
#
# 1. Plot a 2x3 grid showing 6 random training samples — each subplot should show
#    the port image with the expert trajectory overlaid. Title each with the
#    destination block/row.
#
# 2. Compute and plot the distribution of waypoint HEADINGS (theta values) across
#    the dataset. This tells you whether trajectories are predominantly straight,
#    turning left, or turning right. A balanced distribution is important for
#    training an unbiased model.
#
# 3. Plot a heatmap of trajectory ENDPOINTS across the port grid. This shows which
#    destinations are sampled most frequently. Ideally, all blocks should have
```

```
#    roughly equal representation.
#
# Thought questions (answer in a text cell below):
# Q1: Are any trajectory heading directions underrepresented? How might this
#     bias the model's turning behavior?
# Q2: If certain port blocks have more endpoint samples, would the model
#     navigate to those blocks more accurately? What data balancing strategy
#     would you use?
# Q3: How does the trajectory length distribution relate to the distance between
#     blocks? Are short trajectories (nearby destinations) overrepresented?

# YOUR CODE HERE
```

3.3 Baseline Approach

```
# --- Baseline: Straight-Line Planner ---
# The simplest baseline: draw a straight line from start to destination,
# ignoring all obstacles. This represents PacificTow's "no ML" approach.

class StraightLineBaseline:
    """Predict a straight-line trajectory from current position to destination."""

    def predict(self, image: torch.Tensor, instruction: torch.Tensor,
                egomotion: torch.Tensor) -> torch.Tensor:
        """
        Extract destination from the image (channel 2 = destination marker)
        and plan a straight line to it.
        """
        # Find destination from image channel 2
        dest_channel = image[2]  # (128, 128)
        dest_mask = (dest_channel > 0.5)
        if dest_mask.sum() == 0:
            # No destination visible — go straight
            trajectory = torch.zeros(NUM_WAYPOINTS, 3)
            trajectory[:, 1] = torch.linspace(0, 0.3, NUM_WAYPOINTS)
            return trajectory

        # Find center of destination marker
        ys, xs = torch.where(dest_mask)
        dest_y = ys.float().mean().item()
        dest_x = xs.float().mean().item()

        # Current position is center of image
        cy, cx = PORT_SIZE / 2, PORT_SIZE / 2

        # Generate straight-line waypoints (normalized)
        traj_x = torch.linspace(0, (dest_x - cx) / (PORT_SIZE / 2), NUM_WAYPOINTS)
        traj_y = torch.linspace(0, (dest_y - cy) / (PORT_SIZE / 2), NUM_WAYPOINTS)
        theta = torch.atan2(traj_y[-1] - traj_y[0], traj_x[-1] - traj_x[0]).expand(NUM_WAYPOINTS)

        trajectory = torch.stack([traj_x, traj_y, theta], dim=-1)
        return trajectory
```

```
# --- Evaluate Baseline ---

def compute_metrics(pred_traj: torch.Tensor, gt_traj: torch.Tensor) -> Dict[str, float]:
    """Compute ADE, FDE, and smoothness metrics."""
    # Average Displacement Error
    displacement = torch.sqrt(((pred_traj[:, :2] - gt_traj[:, :2]) ** 2).sum(dim=-1))
    ade = displacement.mean().item()

    # Final Displacement Error
    fde = displacement[-1].item()

    # Smoothness (jerk = third derivative of position)
    dt = 0.1  # 10Hz
    velocity = torch.diff(pred_traj[:, :2], dim=0) / dt
    acceleration = torch.diff(velocity, dim=0) / dt
    jerk = torch.diff(acceleration, dim=0) / dt
    smoothness = torch.sqrt((jerk ** 2).sum(dim=-1)).mean().item()

    return {"ADE": ade, "FDE": fde, "Jerk": smoothness}
```

```
# Evaluate straight-line baseline
baseline = StraightLineBaseline()
baseline_metrics = {"ADE": [], "FDE": [], "Jerk": []}

for i in range(min(200, len(val_dataset))):
    image, instruction, egomotion, gt_traj = val_dataset[i]
    pred = baseline.predict(image, instruction, egomotion)
    metrics = compute_metrics(pred, gt_traj)
    for k, v in metrics.items():
        baseline_metrics[k].append(v)

print("=== Straight-Line Baseline Results ===")
for k in baseline_metrics:
    vals = baseline_metrics[k]
    print(f"  {k}: {np.mean(vals):.4f} +/- {np.std(vals):.4f}")
```

TODO: Improved Baseline

```
# TODO 3: Implement an Improved Baseline

class ObstacleAwareBaseline:
    """
    A baseline that plans a trajectory toward the destination while avoiding
    obstacles visible in the image.

    This represents what a traditional (non-ML) planner would do: extract
    obstacle positions from the sensor data and plan around them using
    geometric rules. It still cannot understand language instructions, but
    it should produce collision-free trajectories.

    The improvement over StraightLineBaseline demonstrates the value of
    obstacle awareness — but the inability to process language instructions
    shows why a VLA approach is needed.

    Args:
        safety_margin: Minimum distance (in grid cells) to maintain from obstacles

    Hints:
        Step 1: Extract the destination from image channel 2 (same as StraightLineBaseline)
        Step 2: Extract obstacle positions from image channel 0 (containers) and
                channel 1 (workers)
        Step 3: Generate a straight-line trajectory to the destination
        Step 4: For each waypoint, check if it is within safety_margin of any obstacle.
                If so, shift it perpendicular to the trajectory direction by safety_margin.
        Step 5: Smooth the adjusted trajectory using a simple moving average
        Step 6: Recompute heading angles from the smoothed x,y positions
    """

    def __init__(self, safety_margin: float = 3.0):
        self.safety_margin = safety_margin

    def predict(self, image: torch.Tensor, instruction: torch.Tensor,
                egomotion: torch.Tensor) -> torch.Tensor:
        # YOUR CODE HERE (approximately 25-35 lines)
        pass


# Evaluate your improved baseline
# improved_baseline = ObstacleAwareBaseline()
# ... (run same evaluation loop as above and compare)
```

```
# --- Verification for TODO 3 ---
if ObstacleAwareBaseline().predict(val_dataset[0][0], val_dataset[0][1], val_dataset[0][2]) is not None:
    improved = ObstacleAwareBaseline()
    imp_metrics = {"ADE": [], "FDE": [], "Jerk": []}
    for i in range(min(200, len(val_dataset))):
        image, instruction, egomotion, gt_traj = val_dataset[i]
        pred = improved.predict(image, instruction, egomotion)
        assert pred.shape == (NUM_WAYPOINTS, 3), f"Wrong shape: {pred.shape}"
        m = compute_metrics(pred, gt_traj)
        for k, v in m.items():
```

```
            imp_metrics[k].append(v)

    print("=== Obstacle-Aware Baseline Results ===")
    for k in imp_metrics:
        vals = imp_metrics[k]
        print(f"  {k}: {np.mean(vals):.4f} +/- {np.std(vals):.4f}")
    print("\n(Compare with straight-line baseline above — ADE/FDE should be similar")
    print(" or slightly worse, but the trajectory should avoid obstacles.)")
else:
    print("[SKIP] ObstacleAwareBaseline not yet implemented")
```

3.4 Model Design

We build a simplified VLA with three components: 1. A CNN vision encoder (simpler than ViT for this synthetic domain) 2. A text encoder for dispatcher instructions 3. A diffusion-based trajectory decoder conditioned on both

```
# --- Component 1: Vision Encoder ---

class VisionEncoder(nn.Module):
    """
    Encode the port scene image into a feature vector.

    Architecture: Simple CNN (3 conv layers + global average pooling).
    In a production VLA, this would be a Vision Transformer. We use a CNN
    here for training efficiency on Colab.

    Input: (B, 3, 128, 128) — 3-channel port image
    Output: (B, 256) — scene feature vector
    """

    def __init__(self, feature_dim: int = 256):
        super().__init__()
        self.conv = nn.Sequential(
            nn.Conv2d(3, 32, 5, stride=2, padding=2),
            nn.ReLU(),
            nn.Conv2d(32, 64, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.Conv2d(64, 128, 3, stride=2, padding=1),
            nn.ReLU(),
            nn.AdaptiveAvgPool2d(1),
        )
        self.fc = nn.Linear(128, feature_dim)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        B = x.shape[0]
        h = self.conv(x).view(B, -1)
        return self.fc(h)


# --- Component 2: Text Encoder ---

class TextEncoder(nn.Module):
    """
    Encode dispatcher instructions into a feature vector.

    Architecture: Embedding + 1-layer Transformer + mean pooling.
    In a production VLA, this would be a pre-trained LLM. We use a small
    trainable transformer here.

    Input: (B, max_len) — tokenized instruction
    Output: (B, 256) — instruction feature vector
    """

    def __init__(self, vocab_size: int, embed_dim: int = 128,
                 feature_dim: int = 256, max_len: int = 32):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=0)
        self.pos_embedding = nn.Embedding(max_len, embed_dim)
        encoder_layer = nn.TransformerEncoderLayer(
            d_model=embed_dim, nhead=4, dim_feedforward=256,
```

```
            dropout=0.1, batch_first=True
        )
        self.transformer = nn.TransformerEncoder(encoder_layer, num_layers=1)
        self.fc = nn.Linear(embed_dim, feature_dim)

    def forward(self, tokens: torch.Tensor) -> torch.Tensor:
        B, L = tokens.shape
        positions = torch.arange(L, device=tokens.device).unsqueeze(0).expand(B, -1)
        x = self.embedding(tokens) + self.pos_embedding(positions)
        # Create padding mask
        pad_mask = (tokens == 0)
        x = self.transformer(x, src_key_padding_mask=pad_mask)
        # Mean pool over non-padded positions
        mask = (~pad_mask).float().unsqueeze(-1)
        x = (x * mask).sum(dim=1) / mask.sum(dim=1).clamp(min=1)
        return self.fc(x)
```

```
# --- Component 3: Diffusion Trajectory Decoder ---

class SinusoidalTimestepEmbedding(nn.Module):
    """Sinusoidal positional encoding for diffusion timestep."""

    def __init__(self, dim: int):
        super().__init__()
        self.dim = dim

    def forward(self, t: torch.Tensor) -> torch.Tensor:
        half_dim = self.dim // 2
        emb = math.log(10000) / (half_dim - 1)
        emb = torch.exp(torch.arange(half_dim, device=t.device, dtype=torch.float32) * -emb)
        emb = t.float().unsqueeze(-1) * emb.unsqueeze(0)
        emb = torch.cat([torch.sin(emb), torch.cos(emb)], dim=-1)
        return emb


class TrajectoryDenoiser(nn.Module):
    """
    Predicts noise added to a trajectory, conditioned on:
    - The noisy trajectory itself
    - The diffusion timestep
    - Scene features (from vision encoder)
    - Instruction features (from text encoder)

    This is the core of the diffusion action decoder. The architecture is
    an MLP (matching the article's description of Alpamayo's approach, scaled
    down for our synthetic task).

    Input:
        noisy_traj: (B, 32, 3) — noisy trajectory
        timestep: (B,) — diffusion timestep [0, T)
        condition: (B, 512) — concatenated vision + text features

    Output: (B, 32, 3) — predicted noise
    """

    def __init__(self, traj_dim: int = 32 * 3, cond_dim: int = 512,
                 hidden_dim: int = 512, time_dim: int = 128):
        super().__init__()
        self.time_embed = SinusoidalTimestepEmbedding(time_dim)
        self.time_proj = nn.Linear(time_dim, hidden_dim)
        self.cond_proj = nn.Linear(cond_dim, hidden_dim)

        self.net = nn.Sequential(
            nn.Linear(traj_dim + hidden_dim + hidden_dim, hidden_dim),
            nn.SiLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.SiLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.SiLU(),
            nn.Linear(hidden_dim, traj_dim),
        )

    def forward(self, noisy_traj: torch.Tensor, timestep: torch.Tensor,
                condition: torch.Tensor) -> torch.Tensor:
        B = noisy_traj.shape[0]
```

```
        traj_flat = noisy_traj.view(B, -1)  # (B, 96)
        t_emb = self.time_proj(self.time_embed(timestep))  # (B, hidden)
        c_emb = self.cond_proj(condition)  # (B, hidden)

        x = torch.cat([traj_flat, t_emb, c_emb], dim=-1)
        noise_pred = self.net(x)
        return noise_pred.view(B, 32, 3)
```

```
# --- Noise Schedule ---

class NoiseSchedule:
    """Linear noise schedule for diffusion."""

    def __init__(self, num_steps: int = 100, beta_start: float = 1e-4, beta_end: float = 0.02):
        self.num_steps = num_steps
        self.beta = torch.linspace(beta_start, beta_end, num_steps)
        self.alpha = 1.0 - self.beta
        self.alpha_bar = torch.cumprod(self.alpha, dim=0)
        self.sqrt_alpha_bar = torch.sqrt(self.alpha_bar)
        self.sqrt_one_minus_alpha_bar = torch.sqrt(1.0 - self.alpha_bar)

    def to(self, device):
        self.beta = self.beta.to(device)
        self.alpha = self.alpha.to(device)
        self.alpha_bar = self.alpha_bar.to(device)
        self.sqrt_alpha_bar = self.sqrt_alpha_bar.to(device)
        self.sqrt_one_minus_alpha_bar = self.sqrt_one_minus_alpha_bar.to(device)
        return self


schedule = NoiseSchedule(num_steps=100)
```

TODO: Implement the Full VLA Model

```
# TODO 4: Assemble the Full VLA Model

class PortVLA(nn.Module):
    """
    A simplified Vision-Language-Action model for port terminal tractors.

    This model combines:
    1. A VisionEncoder to process the port scene image
    2. A TextEncoder to process the dispatcher instruction
    3. A TrajectoryDenoiser (diffusion decoder) to generate trajectories

    The forward pass for TRAINING:
    - Encode the image and instruction into feature vectors
    - Concatenate them into a conditioning vector
    - Sample a random timestep and add noise to the expert trajectory
    - Predict the noise using the denoiser
    - Return the predicted noise and actual noise (for loss computation)

    The forward pass for INFERENCE (sampling):
    - Encode image and instruction
    - Start from pure noise
    - Iteratively denoise using the reverse diffusion process
    - Return the generated trajectory

    Args:
        vocab_size: Size of instruction vocabulary
        vision_dim: Output dimension of vision encoder
        text_dim: Output dimension of text encoder
        num_diffusion_steps: Number of diffusion timesteps

    Hints:
        __init__:
            Step 1: Create VisionEncoder(feature_dim=vision_dim)
            Step 2: Create TextEncoder(vocab_size, feature_dim=text_dim)
            Step 3: Create TrajectoryDenoiser(cond_dim=vision_dim + text_dim)
            Step 4: Create NoiseSchedule(num_steps=num_diffusion_steps)

        forward (training mode):
            Step 1: Encode image with vision encoder -> (B, vision_dim)
```

```
            Step 2: Encode instruction with text encoder -> (B, text_dim)
            Step 3: Concatenate vision and text features -> (B, vision_dim + text_dim)
            Step 4: Sample random timesteps: torch.randint(0, T, (B,))
            Step 5: Sample random noise: torch.randn_like(trajectory)
            Step 6: Create noisy trajectory using the forward diffusion formula:
                    noisy = sqrt_alpha_bar[t] * trajectory + sqrt_one_minus_alpha_bar[t] * noise
                    (Remember to reshape the schedule values for broadcasting: [..., None, None])
            Step 7: Predict noise using denoiser(noisy, timesteps, condition)
            Step 8: Return (predicted_noise, actual_noise)

        sample (inference mode — implement as a separate method):
            Step 1: Encode image and instruction, concatenate
            Step 2: Start from pure noise: x = torch.randn(B, 32, 3)
            Step 3: Loop from t = T-1 down to 0:
                    a. Predict noise: eps = denoiser(x, t, condition)
                    b. Compute the denoised mean using the DDPM reverse formula
                    c. If t > 0, add scaled random noise
            Step 4: Return the final denoised trajectory
    """

    def __init__(self, vocab_size: int, vision_dim: int = 256,
                 text_dim: int = 256, num_diffusion_steps: int = 100):
        super().__init__()
        # YOUR CODE HERE (approximately 5-7 lines)
        pass

    def forward(self, image: torch.Tensor, instruction: torch.Tensor,
                trajectory: torch.Tensor):
        """Training forward pass. Returns (predicted_noise, actual_noise)."""
        # YOUR CODE HERE (approximately 10-12 lines)
        pass

    @torch.no_grad()
    def sample(self, image: torch.Tensor, instruction: torch.Tensor,
               num_samples: int = 1) -> torch.Tensor:
        """Generate trajectory samples via reverse diffusion."""
        # YOUR CODE HERE (approximately 15-20 lines)
        pass
```

```
# --- Verification for TODO 4 ---

# Build model (or use reference if TODO not complete)
try:
    model = PortVLA(vocab_size=train_dataset.vocab_size).to(device)

    # Test training forward pass
    test_img = torch.randn(4, 3, 128, 128).to(device)
    test_inst = torch.randint(0, train_dataset.vocab_size, (4, 32)).to(device)
    test_traj = torch.randn(4, 32, 3).to(device)

    pred_noise, actual_noise = model(test_img, test_inst, test_traj)
    assert pred_noise.shape == (4, 32, 3), f"Wrong noise shape: {pred_noise.shape}"
    assert actual_noise.shape == (4, 32, 3), f"Wrong noise shape: {actual_noise.shape}"
    print("[PASS] Training forward pass: correct shapes")

    # Test sampling
    sampled = model.sample(test_img[:1], test_inst[:1], num_samples=1)
    assert sampled.shape == (1, 32, 3), f"Wrong sample shape: {sampled.shape}"
    print("[PASS] Sampling: correct shape")

    total_params = sum(p.numel() for p in model.parameters())
    print(f"Total parameters: {total_params:,}")
except Exception as e:
    print(f"[FAIL] {e}")
    print("Implement PortVLA in TODO 4 above, then re-run this cell.")
```

3.5 Training Strategy

```
# --- Training Configuration ---

BATCH_SIZE = 64
LEARNING_RATE = 3e-4
```

```
NUM_EPOCHS = 30
WEIGHT_DECAY = 1e-4
WARMUP_STEPS = 200

train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True, drop_last=True)
val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE, shuffle=False)
```

TODO: Implement the Training Loop

```
# TODO 5: Implement the Training Loop

def train_vla(model, train_loader, val_loader, num_epochs, lr, weight_decay, warmup_steps):
    """
    Train the VLA model using the diffusion denoising objective.

    The training objective is simple: predict the noise that was added to the
    expert trajectory. The loss is MSE between predicted and actual noise.

    We use AdamW (not Adam) because weight decay helps prevent overfitting on
    our relatively small synthetic dataset. The cosine learning rate schedule
    with warmup provides stable early training (warmup prevents large initial
    gradient updates) and graceful convergence (cosine decay avoids the abrupt
    drops of step schedules).

    Args:
        model: PortVLA model
        train_loader: Training data loader
        val_loader: Validation data loader
        num_epochs: Number of training epochs
        lr: Peak learning rate
        weight_decay: AdamW weight decay
        warmup_steps: Number of linear warmup steps

    Returns:
        Dict with 'train_losses' and 'val_losses' lists

    Hints:
        Step 1: Create AdamW optimizer with the given lr and weight_decay
        Step 2: Create a learning rate scheduler — use
                torch.optim.lr_scheduler.CosineAnnealingLR (T_max = num_epochs * len(train_loader))
                For warmup: manually scale LR for the first warmup_steps
        Step 3: For each epoch:
            a. Set model to train mode
            b. For each batch (image, instruction, egomotion, trajectory):
                - Move all tensors to device
                - Forward pass: pred_noise, actual_noise = model(image, instruction, trajectory)
                - Compute MSE loss: F.mse_loss(pred_noise, actual_noise)
                - Backward pass and optimizer step
                - Apply learning rate warmup if within warmup_steps
                - Step the scheduler
            c. Compute validation loss (model.eval(), no gradients)
            d. Print epoch stats: train loss, val loss, learning rate
        Step 4: Return the loss histories
    """
    # YOUR CODE HERE (approximately 35-45 lines)
    pass
```

```
# --- Run Training ---
# (Uncomment after implementing TODO 5)

# history = train_vla(model, train_loader, val_loader,
#                     num_epochs=NUM_EPOCHS, lr=LEARNING_RATE,
#                     weight_decay=WEIGHT_DECAY, warmup_steps=WARMUP_STEPS)
```

```
# --- Visualize Training Curves ---
# (Uncomment after training completes)

# fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))
#
# ax1.plot(history['train_losses'], label='Train Loss', alpha=0.7)
# ax1.plot(history['val_losses'], label='Val Loss', alpha=0.7)
# ax1.set_xlabel('Epoch')
```

```
# ax1.set_ylabel('MSE Loss')
# ax1.set_title('Training and Validation Loss')
# ax1.legend()
# ax1.grid(True, alpha=0.3)
#
# # Learning rate schedule visualization
# ax2.plot(history.get('lrs', []), alpha=0.7)
# ax2.set_xlabel('Step')
# ax2.set_ylabel('Learning Rate')
# ax2.set_title('Learning Rate Schedule')
# ax2.grid(True, alpha=0.3)
#
# plt.tight_layout()
# plt.show()
```

3.6 Evaluation

```
# --- Quantitative Evaluation ---

def evaluate_model(model, val_loader, num_samples=5):
    """
    Evaluate the VLA model on the validation set.
    For each sample, generate num_samples trajectories and pick the best (lowest ADE).
    This leverages the diffusion model's multimodality — try multiple samples and
    select the one closest to the expert trajectory.
    """
    model.eval()
    all_metrics = {"ADE": [], "FDE": [], "Jerk": []}

    with torch.no_grad():
        for batch_idx, (image, instruction, egomotion, gt_traj) in enumerate(val_loader):
            if batch_idx >= 10:  # Evaluate on 10 batches
                break
            image = image.to(device)
            instruction = instruction.to(device)
            gt_traj = gt_traj.to(device)

            # Generate for first sample in batch
            for i in range(min(4, image.shape[0])):
                best_ade = float('inf')
                best_pred = None

                for _ in range(num_samples):
                    pred = model.sample(image[i:i+1], instruction[i:i+1])
                    m = compute_metrics(pred[0].cpu(), gt_traj[i].cpu())
                    if m["ADE"] < best_ade:
                        best_ade = m["ADE"]
                        best_pred = pred[0].cpu()
                        best_metrics = m

                for k, v in best_metrics.items():
                    all_metrics[k].append(v)

    print("=== VLA Model Evaluation (best of 5 samples) ===")
    for k, vals in all_metrics.items():
        print(f"  {k}: {np.mean(vals):.4f} +/- {np.std(vals):.4f}")

    return all_metrics


# Uncomment after training:
# vla_metrics = evaluate_model(model, val_loader)
```

TODO: Comparative Evaluation

```
# TODO 6: Generate a Comparative Evaluation

# After training, complete the following evaluation:
#
# 1. Run evaluate_model() on your trained VLA
# 2. Collect baseline metrics from Section 3.3
```

```
# 3. Create a bar chart comparing ADE, FDE, and Jerk for:
#    - Straight-Line Baseline
#    - Obstacle-Aware Baseline (your TODO 3)
#    - VLA Model
#    Use grouped bars with error bars showing +/- 1 std deviation.
#    Include a horizontal dashed line at the target threshold for each metric.
#
# 4. Create a 2x3 figure showing 6 validation samples:
#    - Each subplot: port image + ground-truth trajectory (green) + VLA prediction (red)
#    - Title each with ADE for that sample
#
# 5. Answer these thought questions:
#    Q1: Where does the VLA outperform the baselines most significantly? Why?
#    Q2: Are there scenarios where the straight-line baseline is actually better
#        than the VLA? Why might this happen?
#    Q3: How would you expect performance to change if we increased the dataset
#        from 5,000 to 50,000 samples? Which metric would improve most?
#
# YOUR CODE HERE
```

3.7 Error Analysis

```
# TODO 7: Systematic Error Analysis

# After training and evaluation, perform a systematic error analysis:
#
# 1. Identify the 20 validation samples with the HIGHEST ADE (worst predictions).
#    For each, visualize the scene image, ground-truth trajectory, and VLA prediction.
#    Look for patterns: are failures concentrated in specific port areas, long
#    distances, scenes with many workers, or particular instruction types?
#
# 2. Categorize failures into at least 3 categories. Suggested categories:
#    - "Long distance": destination is far away, trajectory accumulates error
#    - "Obstacle avoidance": trajectory clips an obstacle
#    - "Instruction misunderstanding": trajectory goes to wrong destination
#    - "Multimodal confusion": trajectory is a valid path but to a different
#      plausible destination (evidence of mode averaging)
#
# 3. For each category:
#    - Count how many of the top-20 failures fall into it
#    - Propose a specific fix (more data, architectural change, loss modification)
#    - Explain WHY that fix addresses the root cause
#
# 4. Compute the collision rate: what percentage of VLA-generated trajectories
#    pass within 1 grid cell of an obstacle (container or worker)?
#
# YOUR CODE HERE
```

3.8 Scalability and Deployment Considerations

```
# --- Inference Latency Profiling ---

import time

def profile_inference(model, image, instruction, num_runs=20):
    """Profile inference latency by component."""
    model.eval()
    image = image.to(device)
    instruction = instruction.to(device)

    # Warm up
    for _ in range(3):
        _ = model.sample(image, instruction)

    # Profile
    times = []
    for _ in range(num_runs):
        torch.cuda.synchronize() if device.type == 'cuda' else None
        start = time.perf_counter()
        _ = model.sample(image, instruction)
        torch.cuda.synchronize() if device.type == 'cuda' else None
```

```
        times.append((time.perf_counter() - start) * 1000)

    times = np.array(times)
    print(f"=== Inference Latency ({num_runs} runs) ===")
    print(f"  Mean: {times.mean():.1f} ms")
    print(f"  Std:  {times.std():.1f} ms")
    print(f"  P50:  {np.percentile(times, 50):.1f} ms")
    print(f"  P95:  {np.percentile(times, 95):.1f} ms")
    print(f"  P99:  {np.percentile(times, 99):.1f} ms")
    return times

# Uncomment after training:
# test_img = val_dataset[0][0].unsqueeze(0)
# test_inst = val_dataset[0][1].unsqueeze(0)
# latencies = profile_inference(model, test_img, test_inst)
```

```
# TODO 8: DDIM Accelerated Sampling

def ddim_sample(model, image, instruction, num_ddim_steps=10):
    """
    Implement DDIM (Denoising Diffusion Implicit Models) sampling to reduce
    the number of denoising steps from 100 to 10.

    DDIM is deterministic — it skips timesteps by taking larger denoising jumps.
    This is critical for deployment: 100 denoising steps at ~2ms each = 200ms,
    which barely meets the latency budget. 10 DDIM steps = ~20ms, leaving
    headroom for vision and text encoding.

    Args:
        model: Trained PortVLA model
        image: (1, 3, 128, 128) input image
        instruction: (1, 32) tokenized instruction
        num_ddim_steps: Number of DDIM sampling steps (default 10)

    Returns:
        trajectory: (1, 32, 3) generated trajectory

    Hints:
        Step 1: Compute the DDIM timestep subsequence — evenly space num_ddim_steps
                timesteps across [0, T). Use torch.linspace(0, T-1, num_ddim_steps).long()
        Step 2: Encode image and instruction to get the condition vector
        Step 3: Start from noise x = torch.randn(1, 32, 3)
        Step 4: Loop through timesteps in REVERSE order. At each step:
                a. Predict noise: eps = denoiser(x, t, condition)
                b. Compute predicted x0: x0_pred = (x - sqrt(1-alpha_bar_t) * eps) / sqrt(alpha_bar_t)
                c. If not the last step, compute x at the previous timestep:
                   x = sqrt(alpha_bar_prev) * x0_pred + sqrt(1 - alpha_bar_prev) * eps
                   (This is the DDIM update — no stochastic noise added)
                d. If last step, x = x0_pred
        Step 5: Return x
    """
    # YOUR CODE HERE (approximately 20-25 lines)
    pass


# Uncomment after implementing:
# ddim_traj = ddim_sample(model, test_img.to(device), test_inst.to(device))
# print(f"DDIM trajectory shape: {ddim_traj.shape}")
# ddim_latencies = profile_inference_fn(lambda: ddim_sample(model, test_img.to(device), test_inst.to(device)))
```

3.9 Ethical and Regulatory Analysis

```
# TODO 9: Ethical Impact Assessment

# Write a brief (300-500 word) ethical impact assessment for deploying the
# VLA-based autonomous terminal tractor system. Address the following:
#
# 1. WORKER SAFETY
#    - What happens when the model hallucinates an obstacle and brakes
#      unnecessarily? (nuisance stops)
#    - What happens when it fails to detect a real worker? (critical failure)
#    - How should the system handle uncertainty — should it be biased toward
```

```
#      false positives (stopping too often) or false negatives (missing threats)?
#      Justify your answer with the asymmetric cost analysis.
#
# 2. LABOR DISPLACEMENT
#    - 140 human drivers at this terminal would be reduced to 20 supervisors.
#      What is PacificTow's ethical obligation to displaced workers?
#    - Consider: retraining programs, transition period, early retirement
#      options, revenue sharing with the union.
#
# 3. ACCOUNTABILITY
#    - If the autonomous tractor causes an accident, who is liable:
#      PacificTow (the autonomy provider), the terminal operator, or the
#      tractor manufacturer?
#    - How does the Chain-of-Causation reasoning trace help or complicate
#      liability determination?
#
# 4. BIAS AND FAIRNESS
#    - The training data comes from 14 months at one terminal. How might
#      geographic/operational bias affect deployment at other ports?
#    - Are there worker demographics (height, clothing color, mobility aids)
#      that might affect detection rates?
#
# 5. REGULATORY COMPLIANCE
#    - List 3 specific regulations or standards that apply to this deployment
#      (hint: MTSA, OSHA, ISO 3691 for industrial trucks).
#    - For each, state one specific requirement the system must meet.
#
# Write your assessment as a markdown text cell below this code cell.
# This is not a coding exercise — it is a critical thinking exercise.
```

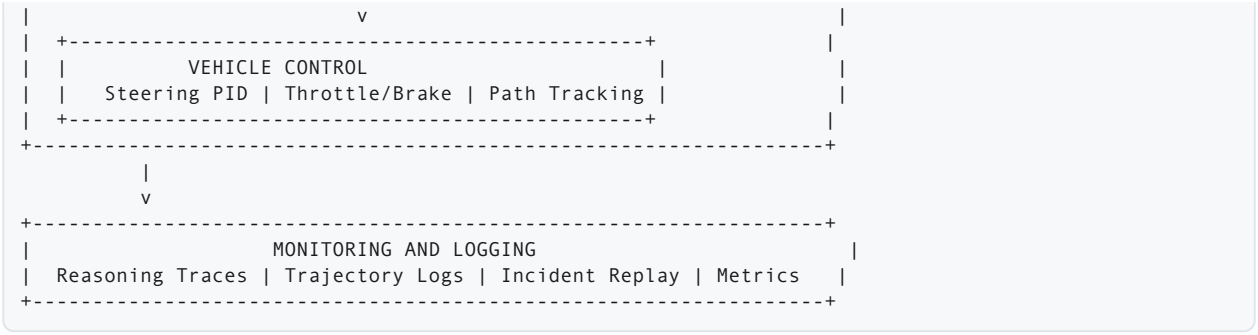
Section 4: Production and System Design Extension

This section describes how PacificTow would deploy the VLA system at production scale. It is written as a system design document for advanced students.

Architecture Overview

The production system has four layers:

```
+-------------------------------------------------------------------+
|                    FLEET MANAGEMENT LAYER                         |
|  Dispatcher Interface | Route Optimizer | Fleet Telemetry         |
+-------------------------------------------------------------------+
          |                          |                     |
          v                          v                     v
+-------------------------------------------------------------------+
|                    AUTONOMY STACK (per tractor)                   |
|                                                                   |
|  +------------------+    +-------------------+                    |
|  | Perception Module|    | Language Module    |                    |
|  | 4x Camera @ 10Hz |--->| Dispatcher Parser  |                    |
|  | LiDAR @ 20Hz     |    | Safety Alert Parser|                    |
|  +--------+---------+    +--------+----------+                    |
|           |                       |                               |
|           v                       v                               |
|  +-------------------------------------------------+              |
|  |          VLA MODEL (Cosmos-VLM + Diffusion)     |              |
|  |   Scene Understanding -> Reasoning -> Trajectory |              |
|  +------------------------+------------------------+              |
|                           |                                       |
|                           v                                       |
|  +-------------------------------------------------+              |
|  |          SAFETY LAYER                           |              |
|  |   Collision Check | Kinematic Limits | E-Stop   |              |
|  +------------------------+------------------------+              |
|                           |                                       |
```



API Design

Trajectory Planning Service (gRPC)

```
service TrajectoryPlanner {
    rpc PlanTrajectory(PlanRequest) returns (PlanResponse);
    rpc StreamTrajectory(PlanRequest) returns (stream TrajectoryUpdate);
}

message PlanRequest {
    repeated CameraFrame cameras = 1;    // 4 cameras x 4 frames
    EgomotionHistory ego_history = 2;     // 16 waypoints
    string instruction = 3;               // Dispatcher text
    float max_speed_mps = 4;              // Speed limit
}

message PlanResponse {
    repeated Waypoint trajectory = 1;     // 32 waypoints
    string reasoning_trace = 2;           // Chain-of-Causation text
    float confidence = 3;                 // Model confidence [0, 1]
    int64 compute_time_ms = 4;            // Inference latency
}

message Waypoint {
    float x = 1;                          // meters, local frame
    float y = 2;
    float theta = 3;                      // radians
    float timestamp = 4;                  // seconds from now
}
```

Fleet Management REST API

Endpoint	Method	Description
`/api/v1/tractors`	GET	List all tractors and their status
`/api/v1/tractors/{id}/trajectory`	GET	Current planned trajectory
`/api/v1/tractors/{id}/reasoning`	GET	Latest reasoning trace
`/api/v1/tractors/{id}/dispatch`	POST	Send new dispatch instruction
`/api/v1/tractors/{id}/estop`	POST	Emergency stop
`/api/v1/incidents`	GET	List safety incidents
`/api/v1/metrics/throughput`	GET	Container moves per hour

Serving Infrastructure

- **On-tractor compute:** NVIDIA Orin (275 TOPS INT8, 32GB memory). Runs the VLA model locally — no cloud dependency for real-time trajectory planning.

- **Model format:** TensorRT-optimized FP16. The diffusion decoder uses INT8 quantization for the denoising MLP, with FP16 for the vision and text encoders (more sensitive to quantization).
- **Dual-cadence architecture:** The VLM backbone (scene understanding + reasoning) runs at 2 Hz. The diffusion decoder (trajectory generation) runs at 10 Hz, reusing the most recent VLM features. This matches Alpamayo's approach of separating slow reasoning from fast action.
- **Fleet server:** A centralized server (4x NVIDIA A100, 128GB RAM) handles fleet optimization, route assignment, and model update distribution. Runs on-premise at the terminal data center for MTSA compliance.

Latency Budget

Component	Latency (P95)	Notes
Camera capture + ISP	5 ms	Hardware pipeline
Vision encoder (4 cameras)	15 ms	CNN on Orin, TensorRT FP16
Text encoder	3 ms	Small transformer, cached if instruction unchanged
VLM reasoning (shared backbone)	80 ms	Runs at 2 Hz, amortized
Diffusion decoder (10 DDIM steps)	25 ms	INT8 MLP on Orin
Safety checks (collision, kinematics)	2 ms	Rule-based, CPU
Vehicle control command	1 ms	CAN bus write
Total (warm path)	**51 ms**	Excluding VLM (uses cached features)
Total (cold path)	**131 ms**	Including VLM reasoning

The 200ms budget is comfortably met. The warm path (10 Hz trajectory updates with cached VLM features) runs at 51 ms, and the cold path (full VLM re-reasoning at 2 Hz) runs at 131 ms.

Monitoring

Real-time dashboard metrics: - Trajectory ADE vs. recorded expert paths (rolling 1-hour window) - Collision proximity events (any trajectory waypoint within 1m of obstacle) - E-stop frequency (target: < 1 per tractor per shift) - Container moves per hour (throughput) - Instruction parsing success rate - Inference latency P50/P95/P99 - Model confidence distribution

Alerting thresholds: - ADE > 0.5m sustained for > 5 minutes: WARN - Collision proximity < 0.3m: CRITICAL, trigger e-stop review - Inference latency P95 > 180ms: WARN (approaching budget) - Model confidence < 0.3 on > 10% of inferences: WARN (possible domain shift) - Any e-stop triggered: INCIDENT, automatic logging + human review

Model Drift Detection

Port environments change gradually (new container block layouts, seasonal weather shifts, new types of equipment) and suddenly (construction, berth reassignment, holiday surge traffic). The system monitors for both:

Gradual drift: Track the distribution of VLM reasoning tokens over 7-day rolling windows. A shift in the frequency of safety-related tokens ("worker", "obstruction", "reroute") may indicate changing port conditions. Statistical test: Kolmogorov-Smirnov test on the token frequency distribution, with a significance threshold of $p < 0.01$.

Sudden drift: Monitor the trajectory confidence score (diffusion decoder's likelihood estimate). A sustained drop in confidence below the 10th percentile of the training distribution triggers an immediate review. This catches scenarios where the model encounters conditions far outside its training data — for example, a new type of construction barrier it has never seen.

Response: When drift is detected, the system: (1) logs all affected trajectories for human review, (2) increases the safety margin in the collision check layer, (3) flags the need for a model update to the ML team.

Model Versioning

- Models are versioned using semantic versioning: `vla-port-v{major}.{minor}.{patch}`
- Major: architecture change (e.g., upgrading vision encoder)
- Minor: significant retraining (e.g., adding new terminal data)
- Patch: fine-tuning or calibration update
- Model artifacts stored in an on-premise model registry (MLflow) with full lineage tracking: training data snapshot, hyperparameters, evaluation metrics, approval sign-off.
- Rollback: Each tractor stores the current and previous model version. If the new model triggers > 3 e-stops in the first 4 hours of deployment, automatic rollback to the previous version occurs.

A/B Testing

- **Traffic splitting:** In a fleet of 20 autonomous tractors, 15 run the production model (control) and 5 run the candidate model (treatment).
- **Primary metric:** Container moves per hour per tractor (throughput).
- **Guardrail metrics:** Collision proximity events, e-stop frequency, instruction parsing failures. If any guardrail metric degrades by > 10% relative to control, the test is automatically halted.
- **Statistical rigor:** Two-sample t-test on throughput with alpha = 0.05. Minimum test duration: 7 days (to capture weekday/weekend variation). Minimum sample size: 500 container moves per group.
- **Rollout:** If the candidate passes A/B testing, gradual rollout: 25% of fleet for 3 days, then 50% for 3 days, then 100%.

CI/CD for ML

Training pipeline (runs weekly or on-demand): 1. Data ingestion: New driving data from the past week is validated (format checks, outlier detection, labeling quality). 2. Dataset versioning: DVC (Data Version Control) snapshots the training data. 3. Training: 8x A100 for 72 hours. Hyperparameters managed by Weights & Biases sweep. 4. Evaluation gate: The trained model must pass on a held-out test set: - ADE < 0.3m - FDE < 0.8m - Collision rate = 0.0% - Instruction accuracy > 95% - Inference latency < 200ms on Orin (tested via TensorRT export) 5. Human review: ML lead reviews reasoning traces on 50 randomly sampled scenarios. 6. Staging: Model deployed to 2 tractors in a fenced staging area for 24 hours of autonomous operation before entering A/B testing.

Cost Analysis

Training costs (per model version): - 8x A100 GPUs for 72 hours at USD 2.00/GPU-hour = USD 1,152 - Data processing and storage: USD 200/month - Weights & Biases tracking: USD 50/month - Total per training run: ~USD 1,400

Inference costs (per tractor): - NVIDIA Orin module: USD 1,500 one-time hardware cost - Power consumption: 60W continuous = ~USD 50/year electricity - TensorRT optimization engineering: amortized across fleet

Fleet server (centralized): - 4x A100 server: USD 120,000 one-time or USD 3,000/month cloud equivalent - Runs fleet optimization, not inference — could be downsized to 2x A100

Total annual cost for 20-tractor fleet: - Hardware (amortized over 5 years): USD 36,000/year - Training and data pipeline: USD 20,000/year - Fleet server: USD 36,000/year - Engineering team (3 ML engineers maintaining the system): USD 600,000/year - **Total: ~USD 692,000/year**

ROI comparison: The 20-tractor autonomous fleet replaces 120 human drivers (saving USD 10.8M/year in labor) at a cost of USD 692K/year in ML operations. Net savings: USD 10.1M/year. Payback period on initial USD 2M development investment: approximately 2.4 months.