Case Study: Dream-Trained Navigation for Autonomous Warehouse Robots

Using World Models to Eliminate Real-World Training Collisions at FleetPath Robotics

---

Section 1: Industry Context and Business Problem

Industry: Autonomous Logistics and Warehouse Automation

The global warehouse automation market is valued at USD 23 billion and is projected to reach USD 41 billion by 2027. E-commerce growth has pushed fulfillment centers to process 3-5x more orders per day than a decade ago, and labor shortages have made autonomous mobile robots (AMRs) a strategic necessity. Companies like Amazon (Kiva/Proteus), Locus Robotics, and 6 River Systems have deployed tens of thousands of robots across fulfillment networks.

The core technical challenge is not building the robots — it is teaching them to navigate. Warehouses are dynamic, partially observable environments where human workers, forklifts, pallets, and other robots create constantly changing obstacle configurations. A navigation policy that works in one warehouse layout may fail entirely in another.

Company Profile: FleetPath Robotics

- **Founded**: 2020, Austin, Texas
- **Team**: 85 employees (22 ML engineers, 18 robotics engineers, 12 operations)
- **Funding**: Series B, USD 42M raised (lead: Eclipse Ventures)
- **Product**: Fleet of autonomous mobile robots (AMRs) for e-commerce fulfillment centers. Each robot carries a top-mounted RGB camera providing a bird's-eye view of the warehouse floor, plus wheel encoders for odometry.
- **Deployment**: 200+ robots across 8 fulfillment centers for 3 enterprise clients (a mid-market 3PL, a DTC apparel brand, and a grocery delivery startup)
- **Revenue**: USD 11M ARR, with a pipeline of 14 new warehouse deployments contracted for 2025

Business Challenge

FleetPath's current navigation stack uses **Proximal Policy Optimization (PPO)**, a model-free reinforcement learning algorithm. The policy is trained by running the robot in the physical warehouse, collecting millions of state-action-reward tuples, and updating the neural network weights over thousands of gradient steps.

This approach works — but it is painfully slow and expensive to onboard new warehouses.

The numbers tell the story:

- **Training duration per new warehouse**: 6-8 weeks of 16-hour-per-day real-world rollouts
- **Collision rate during training**: ~340 collisions per warehouse onboarding cycle
- **Cost per collision**: USD 2,200 average (robot repairs: USD 800, product damage: USD 600, operational downtime: USD 800)
- **Total onboarding cost**: USD 748,000 per warehouse in collision damage alone, plus USD 180,000 in engineering labor
- **Training data volume**: 8-12 million environment steps per layout

FleetPath has 14 new warehouses to onboard in 2025. At the current rate, that is USD 13M in onboarding costs and a minimum of 18 months of sequential rollout — far exceeding the 2-week deployment window their enterprise contracts specify.

Why It Matters

- **Financial**: USD 13M in projected onboarding costs threatens profitability before the company reaches Series C
- **Competitive**: Rivals offering faster deployment windows are winning contracts
- **Safety**: 12% of training collisions involve near-misses with human workers, creating liability exposure
- **Scale**: The current approach does not scale — each new warehouse layout requires training from scratch

Constraints

Constraint	Specification
Compute budget	4x NVIDIA T4 GPUs per warehouse (on-premise edge server)
Latency	Navigation decisions must run at 10 Hz (100ms per inference)
Data availability	2-3 days of human-teleoperated rollout data per new warehouse (~50,000 frames)
Privacy	SOC 2 Type II compliance required; no warehouse imagery may leave the facility
Deployment	Models must run on-robot (NVIDIA Jetson AGX Orin, 32GB)
Team expertise	Team is proficient in PyTorch and standard RL; limited experience with generative models

---

Section 2: Technical Problem Formulation

Problem Type: Model-Based Reinforcement Learning

FleetPath's navigation task is a sequential decision problem: at each timestep, the robot observes its surroundings, chooses a movement action, and receives feedback on whether it is making progress toward its destination without collisions.

Why model-based RL over model-free RL?

Model-free methods (PPO, SAC, DQN) learn a direct mapping from observations to actions through trial and error. They make no attempt to understand *how the world works* — they simply memorize which actions led to good outcomes. This has two consequences:

1. **Sample inefficiency**: The agent must experience every situation many times to learn from it. There is no generalization from understanding dynamics — only from pattern matching on past experience.
2. **No transfer**: A policy trained in Warehouse A has no understanding of *why* certain actions work, so it cannot transfer that knowledge to Warehouse B.

Model-based RL takes a fundamentally different approach. The agent first learns a **world model** — an internal simulator that predicts how the environment evolves in response to actions. Once this model is learned, the agent can train its policy by *imagining* thousands of scenarios (called "dream rollouts") without ever touching the real environment.

Why not classical planning (A*, RRT)? Classical planners require a complete, accurate map of the environment — including the positions and velocities of all dynamic obstacles. In a busy warehouse, this information is never fully available. The environment is **partially observable**: the robot sees only its immediate surroundings through a camera, and human workers move unpredictably.

Why not imitation learning? Imitation learning requires large datasets of expert demonstrations. FleetPath can collect 2-3 days of teleoperation data per warehouse, but this covers only a fraction of the possible scenarios. The agent must generalize to situations it has never seen — exactly what a world model enables through imagination.

Input Specification

At each timestep t, the robot receives:

- **Visual observation** $x_t \in \mathbb{R}^{64\times64\times3}$: A top-down RGB image from the overhead camera, resized to 64x64 pixels. This captures the robot's local surroundings — aisle geometry, nearby obstacles, other robots, and human workers.
- **Previous action** $a_{t-1} \in \mathbb{R}^3$: The action taken at the previous timestep (steering angle, throttle, brake). This provides the agent with proprioceptive context.

Each input modality carries specific information: - The visual observation provides **spatial context**: where am I, what obstacles are nearby, which direction is the aisle going? - The previous action provides **temporal context**: what was I just doing? This helps the world model predict the next state more accurately, since the robot's dynamics depend on its current velocity and heading (which are partially determined by recent actions).

Output Specification

The controller outputs an action $a_t \in \mathbb{R}^3$:

- $a_t[0] \in [-1, 1]$: Steering angle (negative = left, positive = right)
- $a_t[1] \in [0, 1]$: Throttle (0 = no acceleration, 1 = full acceleration)
- $a_t[2] \in [0, 1]$: Brake (0 = no braking, 1 = full braking)

Why continuous actions instead of discrete? Warehouse robots require smooth, precise movements — jerky discrete actions (hard left, hard right) would be unsafe around human workers and could damage carried goods. Continuous control allows fine-grained adjustments.

Why 3-dimensional output? Steering, throttle, and brake are the minimal set of controls needed for a differential-drive robot. Combining throttle and brake into a single dimension would conflate two distinct physical operations (applying motor torque vs. engaging friction brakes) that have different response curves.

Mathematical Foundation

The world model architecture has three components. Before presenting the loss functions, let us understand the mathematical principles that justify each one.

Principle 1: Learned Compression via Variational Inference

The robot receives 12,288-dimensional observations (64 x 64 x 3 pixels) at 10 Hz. Processing raw pixels for sequential prediction is computationally intractable and riddled with irrelevant detail (exact pixel intensities of floor textures, lighting variations). We need a **compressed representation** that preserves task-relevant information.

The Variational Autoencoder (VAE) provides a principled framework for this compression. The key idea from variational inference is:

Given a dataset of observations $\{x_1, x_2, \ldots, x_N\}$, we want to find a latent representation z such that we can both (a) encode any observation into z , and (b) decode z back to the original observation. The VAE does this by optimizing the **Evidence Lower Bound (ELBO)**:

$$\log p(x) \geq \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)] - D_{\mathrm{KL}}(q_\phi(z|x) \| p(z))$$

This inequality is derived from Jensen's inequality applied to the marginal log-likelihood. The left side, $\log p(x)$, is the log-probability of the observation under our generative model — we want

this to be high. Since computing it exactly is intractable (it requires integrating over all possible z), we maximize the right side (the ELBO) instead.

The two terms have intuitive interpretations:

- **Reconstruction term** $\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)]$: "The decoder should be able to reconstruct x from z." If we remove this term, the encoder could map everything to the same point — useless compression.
- **KL regularization term** $D_{\mathrm{KL}}(q_\phi(z|x)\|p(z))$: "The encoder's distribution should stay close to a standard Gaussian $p(z) = \mathcal{N}(0, I)$." If we remove this term, the encoder could memorize each observation as a distinct point in latent space — no generalization, no smooth interpolation between similar scenes.

The **reparameterization trick** makes this optimization tractable with standard backpropagation:

$$z = \mu_\phi(x) + \sigma_\phi(x) \odot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$$

This rewrites the stochastic sampling as a deterministic function of the encoder outputs (μ, σ) and external noise ϵ, allowing gradients to flow through z to the encoder parameters ϕ.

Principle 2: Multimodal Future Prediction via Mixture Density Networks

The future is not deterministic. When a warehouse robot approaches an intersection, a human worker could appear from the left, the right, or not at all. A single Gaussian prediction would average these possibilities, predicting the worker is "half-visible from both sides" — a state that never actually occurs.

The **Mixture Density Network (MDN)** addresses this by predicting a weighted mixture of K Gaussian components:

$$P(z_{t+1}|z_t, a_t, h_t) = \sum_{i=1}^{K} \pi_i \cdot \mathcal{N}(z_{t+1} \mid \mu_i, \sigma_i^2 I)$$

where: - π_i is the mixing coefficient for component i (with $\sum_i \pi_i = 1$, enforced by softmax) - μ_i is the predicted mean of future i - σ_i is the predicted standard deviation of future i - h_t is the LSTM hidden state encoding the history of observations and actions

The mixing coefficients are computed via softmax to form a valid probability distribution. This naturally captures multimodal futures: if there are two likely outcomes, two components will have high π values; if the future is predictable, one component will dominate.

The MDN is trained by maximizing the log-likelihood of the actual next latent state under the predicted mixture:

$$\mathcal{L}_{\mathrm{MDN}} = -\sum_t \log\left(\sum_{i=1}^{K} \pi_i \cdot \mathcal{N}(z_{t+1}^{\mathrm{actual}} \mid \mu_i, \sigma_i^2 I)\right)$$

Principle 3: Evolutionary Optimization for the Controller

The controller is a single linear layer: $a_t = W_c[z_t; h_t] + b_c$. With 32-dimensional z_t and 256-dimensional h_t, this yields $(32 + 256) \times 3 + 3 = 867$ parameters.

Why not train the controller with gradient descent? Two reasons:

1. **Credit assignment through long rollouts**: The controller's actions affect future states through the world model, creating very long computational graphs. Backpropagation through hundreds of dream steps suffers from vanishing/exploding gradients.
2. **Simplicity**: With only 867 parameters, derivative-free optimization is tractable and avoids the complexity of differentiating through the stochastic MDN sampling.

CMA-ES (Covariance Matrix Adaptation Evolution Strategy) maintains a multivariate Gaussian distribution over the controller parameters and iteratively updates its mean and covariance based on the fitness (cumulative reward) of sampled controllers. It requires no gradient computation and naturally handles noisy, non-differentiable objectives.

Loss Function

The total training procedure involves three losses, applied in sequence:

Loss 1 — VAE Loss (trains V):

$$\mathcal{L}_{\text{VAE}} = \underbrace{\|x - \hat{x}\|^2}_{\text{Reconstruction}} + \underbrace{\beta \cdot D_{\text{KL}}(q_\phi(z|x) \| \mathcal{N}(0, I))}_{\text{Regularization}}$$

- **Reconstruction term**: Mean squared error between input and reconstructed image. Removing this term would mean the VAE never learns to encode useful information — the latent codes would be random noise.
- **KL term**: Keeps the latent space smooth and interpolable. Removing this term causes "posterior collapse" — the encoder memorizes each image as a distinct spike, and similar warehouse scenes map to distant latent codes. The coefficient β controls the trade-off; $\beta = 1$ gives the standard VAE, $\beta < 1$ allows sharper reconstructions at the cost of less regular latent spaces.

Loss 2 — MDN-RNN Loss (trains M):

$$\mathcal{L}_{\text{MDN}} = -\frac{1}{T} \sum_{t=1}^{T} \log \left(\sum_{i=1}^{K} \pi_i^{(t)} \cdot \mathcal{N}(z_{t+1} \mid \mu_i^{(t)}, (\sigma_i^{(t)})^2 I) \right)$$

- This is the negative log-likelihood of the actual next latent state under the predicted mixture distribution.
- Removing any of the K Gaussian components reduces the model's ability to capture multimodal transitions. In practice, $K = 5$ provides a good balance between expressiveness and computational cost.

- If we used $K = 1$ (a single Gaussian), the model would predict the *average* of all possible futures — blurring together distinct outcomes and making the dream world unrealistic.

Loss 3 — Controller Fitness (trains C via CMA-ES):

$$F(\theta_c) = \mathbb{E}_{\text{dream}}\left[\sum_{t=0}^{T} r_t\right]$$

- The fitness function is the expected cumulative reward over dream rollouts. CMA-ES maximizes this by evolving populations of controller parameters θ_c.
- This is not a differentiable loss — it is evaluated by running the controller inside the learned world model and observing the total reward.

Evaluation Metrics

Metric	Target	Rationale
Average episode reward	> 850 (out of 1000)	Primary measure of navigation quality; 900+ indicates human-level in CarRacing
Collision rate	< 2% of timesteps	Safety-critical for warehouse deployment
Sample efficiency	< 50,000 real environment steps	Current model-free baseline requires 8-12M steps
Dream fidelity (FID)	< 50	Measures how realistic the dreamed frames are; low FID indicates the world model captures real dynamics
Inference latency	< 100ms per action	Must run at 10 Hz for real-time navigation

Baseline

Naive baseline: A rule-based proportional controller that steers toward the center of the detected road/aisle using simple color segmentation. This achieves an average reward of ~200-300 on CarRacing — it can navigate gentle curves but fails on sharp turns and has no ability to anticipate upcoming obstacles.

Model-free baseline: PPO trained for 10M environment steps achieves an average reward of ~850-900 but requires 200x more real-world interaction than the world model approach.

The world model must match the model-free baseline's performance (reward > 850) while using < 1% of the real-world training data.

Why World Models Are the Right Approach

The world model architecture directly addresses FleetPath's three core challenges:

1. **Sample efficiency**: The VAE and MDN-RNN learn the environment's dynamics from ~50,000 real frames. The controller then trains for millions of steps *inside the dream* — no

real-world interaction needed. This is the fundamental principle of model-based RL: learn the rules of the game, then practice in imagination.

2. **Safety**: Zero collisions during controller training because all training happens in the learned world model. The only real-world data collection uses a slow, conservative teleoperation policy.

3. **Transferability**: Once the VAE learns a general visual representation of warehouse elements (aisles, obstacles, floor markings), only the MDN-RNN and controller need to be fine-tuned for new layouts — with far less data than training from scratch.

Technical Constraints

Parameter	Value
VAE latent dimension	32
MDN-RNN hidden dimension	256
Number of Gaussian components (K)	5
Controller parameters	867
Training compute	1x T4 GPU, < 90 minutes total
Inference compute	CPU-feasible (linear controller)
Real data budget	10,000 rollout frames (random policy)

Section 3: Implementation Notebook Structure

This section outlines a Google Colab notebook that takes the student from raw data collection to a fully trained world model agent. The notebook uses the CarRacing-v2 environment from OpenAI Gymnasium as a stand-in for FleetPath's warehouse navigation task.

3.1 Environment Setup and Data Collection

Context: Before building a world model, we need training data. In FleetPath's scenario, this corresponds to 2-3 days of teleoperated rollouts in a new warehouse. In our Colab notebook, we collect rollouts from CarRacing-v2 using a random policy.

What the student will do: Install dependencies, initialize the environment, and collect a dataset of observation sequences by running random rollouts.

Provided code: - Environment initialization and frame preprocessing (resize to 64x64, normalize to [0, 1]) - Random rollout collection loop

TODO 1 — Implement the data collection pipeline:

```
def collect_rollouts(env, n_rollouts=100, max_steps=300):
    """
    Collect rollout data from the environment using a random policy.

    Args:
        env: Gymnasium environment instance
        n_rollouts: Number of episodes to collect
        max_steps: Maximum steps per episode

    Returns:
        observations: List of arrays, each of shape (T, 64, 64, 3)
        actions: List of arrays, each of shape (T, 3)

    Hints:
        1. For each rollout, reset the environment and collect frames
        2. Sample random actions using env.action_space.sample()
        3. Preprocess each frame: resize to 64x64 and normalize pixel
           values to [0, 1]
        4. Store observations and actions for each episode separately
        5. Aim for ~10,000 total frames across all rollouts
    """
    # TODO: Implement this function
    pass
```

Verification: The function should return lists where the total number of frames across all rollouts is approximately 10,000. Each observation frame should have shape (64, 64, 3) and values in [0, 1].

Thought questions: - Why do we use a random policy for data collection instead of an expert policy? - How does the quality of the collected data affect the world model's accuracy? - In FleetPath's scenario, why is teleoperation preferred over random exploration for initial data collection?

3.2 Exploratory Data Analysis

Context: Before training any model, we need to understand the structure of our data. This mirrors the data audit that FleetPath's ML team would conduct on teleoperation logs.

What the student will do: Visualize sample frames, analyze action distributions, and inspect frame-to-frame differences.

TODO 2 — Visualize and analyze the collected data:

```
def analyze_rollout_data(observations, actions):
    """
    Perform exploratory data analysis on the collected rollout data.

    Args:
        observations: List of observation arrays from collect_rollouts
        actions: List of action arrays from collect_rollouts

    Tasks:
        1. Plot a grid of 16 sample frames from different rollouts
           to visualize the variety of scenes
        2. Plot histograms of each action dimension (steering, throttle,
           brake) across the entire dataset
        3. Compute and plot the mean pixel difference between consecutive
           frames — this tells us how quickly the scene changes
        4. Report summary statistics: total frames, mean episode length,
           action value ranges

    Hints:
        - Use matplotlib with a 4x4 subplot grid for the frame samples
```

```
        - For consecutive frame differences, compute
          np.mean(np.abs(obs[t+1] - obs[t])) for each t
        - The action distribution will be uniform (since we used a random
          policy) — note this and consider how it would differ with an
          expert policy
    """
    # TODO: Implement this function
    pass
```

Thought questions: - What does the frame difference distribution tell us about the temporal dynamics of the environment? - If the action distribution is uniform, does that mean every region of the state space is equally well-covered? Why or why not? - What biases might exist in FleetPath's teleoperation data that a random policy would not have?

3.3 Baseline: Rule-Based Proportional Controller

Context: Before building a complex world model, we need a baseline to compare against. FleetPath's warehouse robots originally used a PID controller that steered toward detected aisle centerlines. We implement a simpler version: a rule-based controller that steers based on the position of road pixels.

What the student will do: Implement a simple baseline controller and evaluate its performance.

TODO 3 — Implement the rule-based baseline:

```
def rule_based_controller(observation):
    """
    A simple baseline controller that steers based on road position.

    Args:
        observation: A single frame of shape (64, 64, 3), values in [0, 1]

    Returns:
        action: numpy array of shape (3,) — [steering, throttle, brake]

    Algorithm:
        1. Extract the bottom third of the image (where the road is
           closest to the car)
        2. Create a binary mask of road pixels (road is gray/dark,
           grass is green)
        3. Compute the centroid of the road pixels
        4. Set steering proportional to how far the centroid is from
           the image center
        5. Set throttle to a constant value (e.g., 0.3)
        6. Set brake to 0

    Hints:
        - Road pixels have similar R, G, B values (gray) while grass
          pixels have high G relative to R and B
        - A simple threshold: pixel is road if
          abs(R - G) < 0.1 and abs(G - B) < 0.1
        - Steering = K_p * (centroid_x - image_center_x) / image_width
          where K_p is a proportional gain (~0.5 to 1.0)
    """
    # TODO: Implement this function
    pass


def evaluate_controller(env, controller_fn, n_episodes=5, max_steps=1000):
    """
    Evaluate a controller function on the environment.

    Args:
        env: Gymnasium environment instance
```

```
        controller_fn: Function that takes an observation and returns
                       an action
        n_episodes: Number of evaluation episodes
        max_steps: Maximum steps per episode

    Returns:
        mean_reward: Average total reward across episodes
        std_reward: Standard deviation of rewards
        episode_rewards: List of per-episode rewards

    Hints:
        1. For each episode, reset the environment
        2. At each step, preprocess the observation and call controller_fn
        3. Accumulate the reward
        4. Return statistics across episodes
    """
    # TODO: Implement this function
    pass
```

Verification: The rule-based controller should achieve an average reward between 200 and 400. If it scores below 100, the road detection logic likely has a bug. If it scores above 500, double-check that the steering gain is not unrealistically tuned.

Thought questions: - Why does the rule-based controller fail on sharp turns? - What information is the rule-based controller missing that a world model agent would have? - In what warehouse scenarios would a simple PID controller actually be sufficient?

3.4 Model Design: The World Model Architecture

Context: Now we build the three-component world model: Vision (VAE), Memory (MDN-RNN), and Controller. This is the core of the case study, directly implementing the architecture from Ha and Schmidhuber (2018).

What the student will do: Implement the VAE, MDN-RNN, and Controller modules in PyTorch.

TODO 4a — Implement the VAE:

```
class VAE(nn.Module):
    """
    Variational Autoencoder for visual compression.

    Compresses 64x64x3 images into 32-dimensional latent codes.

    Architecture:
        Encoder: 4 convolutional layers (stride 2) reducing spatial
                 dimensions 64 -> 30 -> 14 -> 6 -> 2, followed by
                 two linear heads for mu and logvar
        Decoder: Linear layer expanding to 2x2x256, followed by 4
                 transposed convolutional layers restoring to 64x64x3

    Key concepts to implement:
        - The encoder outputs parameters of a distribution (mu, logvar),
          NOT a deterministic code
        - The reparameterization trick: z = mu + std * epsilon
        - The decoder uses transposed convolutions (the "reverse" of
          convolutions)
    """

    def __init__(self, latent_dim=32):
        super().__init__()
        # TODO: Define encoder layers
        # Hint: Use nn.Conv2d with kernel_size=4, stride=2
        # Layer sizes: 3->32->64->128->256 channels
```

```
        # After convolutions, flatten and project to mu and logvar

        # TODO: Define decoder layers
        # Hint: Use nn.ConvTranspose2d to reverse the encoder
        # First project from latent_dim to 256*2*2 with a linear layer
        # Then use transposed convolutions: 256->128->64->32->3 channels
        # Use kernel sizes [5, 5, 6, 6] and stride 2 for the decoder
        # Final activation should be Sigmoid (pixel values in [0,1])
        pass

    def encode(self, x):
        """
        Encode an image to latent distribution parameters.

        Args:
            x: Image tensor of shape (batch, 3, 64, 64)

        Returns:
            mu: Mean of shape (batch, latent_dim)
            logvar: Log-variance of shape (batch, latent_dim)
        """
        # TODO: Pass x through encoder convolutions, flatten,
        # project to mu and logvar
        pass

    def reparameterize(self, mu, logvar):
        """
        Sample from the latent distribution using the
        reparameterization trick.

        Args:
            mu: Mean of shape (batch, latent_dim)
            logvar: Log-variance of shape (batch, latent_dim)

        Returns:
            z: Sampled latent code of shape (batch, latent_dim)

        Key insight: z = mu + std * epsilon where epsilon ~ N(0, I)
        This makes the sampling differentiable w.r.t. mu and logvar.
        std = exp(0.5 * logvar) because logvar = log(sigma^2),
        so exp(0.5 * logvar) = sigma.
        """
        # TODO: Implement the reparameterization trick
        pass

    def decode(self, z):
        """
        Decode a latent code back to an image.

        Args:
            z: Latent code of shape (batch, latent_dim)

        Returns:
            x_recon: Reconstructed image of shape (batch, 3, 64, 64)
        """
        # TODO: Project z to spatial dimensions, apply transposed
        # convolutions
        pass

    def forward(self, x):
        """Full forward pass: encode, sample, decode."""
        mu, logvar = self.encode(x)
        z = self.reparameterize(mu, logvar)
        return self.decode(z), mu, logvar
```

TODO 4b — Implement the MDN-RNN:

```
class MDNRNN(nn.Module):
    """
    Mixture Density Network RNN for dynamics prediction.

    Takes a sequence of (latent_code, action) pairs and predicts
    the distribution of the next latent code as a mixture of
    Gaussians.
```

```
    Architecture:
        - LSTM with hidden_dim=256
        - Three linear heads projecting hidden state to:
          pi (mixing coefficients), mu (means), sigma (std devs)
          for K=5 Gaussian components

    Key concepts:
        - The LSTM hidden state h_t serves as the agent's memory
        - The MDN head outputs parameters for K Gaussians, allowing
          multimodal predictions (multiple possible futures)
        - pi values are passed through softmax to form a valid
          probability distribution
        - sigma values are exponentiated to ensure positivity
    """

    def __init__(self, latent_dim=32, action_dim=3,
                 hidden_dim=256, n_gaussians=5):
        super().__init__()
        # TODO: Define the LSTM layer
        # Input size: latent_dim + action_dim (concatenated)
        # Hidden size: hidden_dim
        # Use batch_first=True

        # TODO: Define three linear heads for pi, mu, sigma
        # Each outputs: latent_dim * n_gaussians values
        # (one set of Gaussian parameters per latent dimension)
        pass

    def forward(self, z, a, hidden=None):
        """
        Predict the distribution of the next latent state.

        Args:
            z: Latent codes, shape (batch, seq_len, latent_dim)
            a: Actions, shape (batch, seq_len, action_dim)
            hidden: Optional LSTM hidden state tuple

        Returns:
            pi: Mixing coefficients (batch*seq_len, latent_dim, K)
            mu: Means (batch*seq_len, latent_dim, K)
            sigma: Std devs (batch*seq_len, latent_dim, K)
            hidden: Updated LSTM hidden state

        Steps:
            1. Concatenate z and a along the last dimension
            2. Pass through LSTM to get hidden states
            3. Reshape LSTM output for the MDN heads
            4. Apply softmax to pi (mixing coefficients must sum to 1)
            5. Apply exp to sigma (std devs must be positive)
        """
        # TODO: Implement the forward pass
        pass
```

TODO 4c — Implement the Controller:

```
class Controller(nn.Module):
    """
    Simple linear controller.

    Takes the concatenation of the current latent code z_t and the
    LSTM hidden state h_t, and outputs an action.

    This is intentionally simple — all the complexity lives in V
    and M. The controller just makes a linear decision based on
    the already-processed state representation.

    Total parameters: (latent_dim + hidden_dim) * action_dim + action_dim
                    = (32 + 256) * 3 + 3 = 867
    """

    def __init__(self, latent_dim=32, hidden_dim=256, action_dim=3):
        super().__init__()
        # TODO: Define a single linear layer
```

```
        # Input: latent_dim + hidden_dim
        # Output: action_dim
        pass

    def forward(self, z, h):
        """
        Choose an action given the current state representation.

        Args:
            z: Current latent code, shape (batch, latent_dim)
            h: Current LSTM hidden state, shape (batch, hidden_dim)

        Returns:
            action: Action vector, shape (batch, action_dim)
                    Values in [-1, 1] (use tanh activation)
        """
        # TODO: Concatenate z and h, pass through linear layer,
        # apply tanh
        pass
```

Verification: After implementing all three modules: - `VAE()` : should have approximately 4.5M parameters - `MDNRNN()` : should have approximately 2.8M parameters - `Controller()` : should have exactly 867 parameters

```
# Verification cell
vae = VAE()
mdnrnn = MDNRNN()
controller = Controller()
for name, model in [("VAE", vae), ("MDNRNN", mdnrnn),
                    ("Controller", controller)]:
    n_params = sum(p.numel() for p in model.parameters())
    print(f"{name}: {n_params:,} parameters")
assert sum(p.numel() for p in controller.parameters()) == 867, \
    "Controller should have exactly 867 parameters"
```

Thought questions: - Why does the controller use tanh activation but the VAE decoder uses sigmoid? - What would happen if we made the controller a 3-layer MLP instead of a single linear layer? - Why is the MDN-RNN's hidden dimension (256) much larger than the latent dimension (32)?

3.5 Training Strategy

Context: Training happens in three sequential stages, mirroring FleetPath's deployment pipeline: first learn to see, then learn to predict, then learn to act.

What the student will do: Implement training loops for the VAE and MDN-RNN, then use CMA-ES to train the controller in dreams.

TODO 5a — Implement the VAE training loop:

```
def vae_loss(x_recon, x, mu, logvar, beta=1.0):
    """
    Compute the VAE loss: reconstruction + KL divergence.

    Args:
        x_recon: Reconstructed image, shape (batch, 3, 64, 64)
        x: Original image, shape (batch, 3, 64, 64)
        mu: Encoder mean, shape (batch, latent_dim)
        logvar: Encoder log-variance, shape (batch, latent_dim)
        beta: KL weight (beta=1 is standard VAE, beta<1 gives
              sharper reconstructions)

    Returns:
```

```
        loss: Scalar loss value
        recon_loss: Reconstruction component (for logging)
        kl_loss: KL divergence component (for logging)

    Formulas:
        Reconstruction loss = MSE(x_recon, x) summed over pixels
        KL loss = -0.5 * sum(1 + logvar - mu^2 - exp(logvar))

    Hints:
        - The KL formula is the closed-form KL divergence between
          N(mu, sigma^2) and N(0, 1)
        - Sum the KL over the latent dimensions, then average over
          the batch
        - For reconstruction, use F.mse_loss with reduction='sum'
          and divide by batch size
    """
    # TODO: Implement the VAE loss function
    pass


def train_vae(vae, train_data, n_epochs=10, batch_size=64, lr=1e-3):
    """
    Train the VAE on collected observation frames.

    Args:
        vae: VAE model instance
        train_data: Tensor of frames, shape (N, 3, 64, 64)
        n_epochs: Number of training epochs
        batch_size: Batch size
        lr: Learning rate

    Returns:
        vae: Trained VAE model
        losses: List of per-epoch average losses

    Steps:
        1. Create a DataLoader from train_data
        2. Use Adam optimizer with the specified learning rate
        3. For each epoch, iterate over batches:
           a. Forward pass through the VAE
           b. Compute the VAE loss
           c. Backpropagate and update weights
        4. Log reconstruction loss and KL loss separately
        5. Every 2 epochs, visualize 8 original vs. reconstructed
           images side by side

    Hints:
        - Use beta=1.0 for the first pass; if reconstructions are
          too blurry, try beta=0.5
        - Learning rate 1e-3 with Adam works well for this scale
    """
    # TODO: Implement the training loop
    pass
```

TODO 5b — Implement the MDN-RNN training loop:

```
def mdn_loss(pi, mu, sigma, z_next):
    """
    Compute the MDN negative log-likelihood loss.

    Args:
        pi: Mixing coefficients, shape (batch, latent_dim, K)
        mu: Means, shape (batch, latent_dim, K)
        sigma: Std devs, shape (batch, latent_dim, K)
        z_next: Actual next latent state, shape (batch, latent_dim)

    Returns:
        loss: Scalar negative log-likelihood

    Steps:
        1. Expand z_next to match the K dimension:
           z_next_expanded has shape (batch, latent_dim, K)
        2. Compute the Gaussian log-probability for each component:
           log_prob_k = -0.5 * ((z_next - mu_k) / sigma_k)^2
```

```
                        - log(sigma_k) - 0.5 * log(2*pi)
        3. Weight by mixing coefficients (in log space):
           log_weighted = log(pi_k) + log_prob_k
        4. Use logsumexp across the K dimension for numerical
           stability
        5. Sum over latent dimensions and average over the batch

    Hints:
        - Use torch.logsumexp for numerical stability
        - Be careful with the constant: log(2*pi) not log(pi_k)
          -- the mathematical constant pi = 3.14159...
        - sigma should be clamped to a minimum value (e.g., 1e-6)
          to prevent division by zero
    """
    # TODO: Implement the MDN loss function
    pass


def train_mdnrnn(mdnrnn, vae, rollout_obs, rollout_actions,
                 n_epochs=20, seq_len=32, batch_size=16, lr=1e-3):
    """
    Train the MDN-RNN on encoded rollout sequences.

    Args:
        mdnrnn: MDNRNN model instance
        vae: Trained VAE model (frozen, used only for encoding)
        rollout_obs: List of observation arrays from rollouts
        rollout_actions: List of action arrays from rollouts
        n_epochs: Number of training epochs
        seq_len: Sequence length for LSTM training
        batch_size: Batch size
        lr: Learning rate

    Returns:
        mdnrnn: Trained MDNRNN model
        losses: List of per-epoch average losses

    Steps:
        1. Encode all observations into latent codes using the
           trained VAE (with torch.no_grad)
        2. Create training sequences of length seq_len from the
           encoded rollouts
        3. For each epoch, iterate over batches of sequences:
           a. Feed (z_t, a_t) for t=0..T-1 into the MDN-RNN
           b. Compute MDN loss against z_{t+1} for t=0..T-1
           c. Backpropagate and update weights

    Hints:
        - Freeze the VAE during this stage (vae.eval(), no_grad)
        - Use truncated backpropagation: detach hidden states
          between sequences
        - Learning rate 1e-3 with Adam, gradient clipping at 1.0
    """
    # TODO: Implement the training loop
    pass
```

TODO 5c — Implement dream-based controller training with CMA-ES:

```
def dream_rollout(vae, mdnrnn, controller, initial_z, max_steps=200):
    """
    Run a rollout entirely inside the learned world model (the dream).

    Args:
        vae: Trained VAE (used only for initial encoding)
        mdnrnn: Trained MDN-RNN
        controller: Controller to evaluate
        initial_z: Starting latent code, shape (latent_dim,)
        max_steps: Maximum dream steps

    Returns:
        total_reward: Estimated cumulative reward from the dream
        dream_states: List of dreamed latent states

    Steps:
```

```
        1. Initialize h_0 (LSTM hidden state) as zeros
        2. Set z_0 = initial_z
        3. For each step t:
           a. Get action from controller: a_t = C(z_t, h_t)
           b. Predict next state distribution: pi, mu, sigma, h_{t+1}
              = M(z_t, a_t, h_t)
           c. Sample z_{t+1} from the predicted mixture
           d. Estimate reward (use a simple heuristic: reward =
              -|z_t - z_target| or learn a reward predictor)
        4. Return total reward

    Hints:
        - For sampling from the mixture: first choose a component k
          with probability pi_k, then sample from N(mu_k, sigma_k^2)
        - For reward estimation, a simple heuristic is to use the
          magnitude of the action (higher speed = higher reward)
          minus a penalty for large steering (which indicates
          the car is off-track)
        - Keep all tensors on the same device
    """
    # TODO: Implement dream rollouts
    pass


def train_controller_cmaes(vae, mdnrnn, controller,
                           initial_observations, n_generations=50,
                           population_size=32, max_dream_steps=200):
    """
    Train the controller using CMA-ES inside the learned dream.

    Args:
        vae: Trained VAE
        mdnrnn: Trained MDN-RNN
        controller: Controller to train
        initial_observations: Real observations to seed dreams
        n_generations: Number of CMA-ES generations
        population_size: Number of controllers per generation
        max_dream_steps: Steps per dream rollout

    Returns:
        controller: Trained controller
        fitness_history: List of best fitness per generation

    Steps:
        1. Extract initial controller parameters as a flat vector
        2. Initialize CMA-ES with these parameters
           (use cma library or implement simple version)
        3. For each generation:
           a. Sample population_size parameter vectors
           b. For each parameter vector:
              - Load parameters into the controller
              - Run dream_rollout from a random initial observation
              - Record the fitness (total reward)
           c. Update the CMA-ES distribution based on fitness
           d. Log the best fitness
        4. Load the best parameters into the controller

    Hints:
        - Install cma: pip install cma
        - For a simpler alternative, implement a basic evolutionary
          strategy: sample from N(mean, sigma*I), keep top 25%,
          update mean to the average of the top 25%
        - Use multiple dream rollouts per controller and average
          the fitness for more stable evaluation
    """
    # TODO: Implement CMA-ES training
    pass
```

Thought questions: - Why do we train V, M, and C sequentially rather than end-to-end? - What happens if the world model (M) is inaccurate? How would this affect the controller? - The Dreamer algorithm (2020) replaces CMA-ES with backpropagation through the dream. What are the advantages and disadvantages of each approach?

3.6 Evaluation

Context: After training, we evaluate the world model agent in the *real* environment (not the dream) and compare against the rule-based baseline and a model-free reference.

What the student will do: Evaluate the trained agent, visualize its behavior, and compare with baselines.

TODO 6 — Evaluate the trained world model agent:

```
def evaluate_world_model_agent(env, vae, mdnrnn, controller,
                               n_episodes=10, max_steps=1000,
                               render=False):
    """
    Evaluate the trained world model agent in the real environment.

    Args:
        env: Gymnasium environment instance
        vae: Trained VAE
        mdnrnn: Trained MDN-RNN
        controller: Trained Controller
        n_episodes: Number of evaluation episodes
        max_steps: Maximum steps per episode
        render: Whether to save rendered frames

    Returns:
        results: Dictionary containing:
            - mean_reward: Average reward across episodes
            - std_reward: Standard deviation
            - episode_rewards: List of per-episode rewards
            - episode_lengths: List of episode lengths
            - frames: List of rendered frames (if render=True)

    Steps:
        1. For each episode:
           a. Reset environment, get initial observation
           b. Encode initial observation with VAE: z_0 = V(x_0)
           c. Initialize LSTM hidden state: h_0 = zeros
           d. At each step:
              - Get action: a_t = C(z_t, h_t)
              - Step the REAL environment: x_{t+1}, r_t = env.step(a_t)
              - Encode new observation: z_{t+1} = V(x_{t+1})
              - Update LSTM: _, _, _, h_{t+1} = M(z_t, a_t, h_t)
              - Accumulate reward
           e. Record episode reward and length
        2. Compute and return statistics

    Hints:
        - Set all models to eval mode and use torch.no_grad()
        - The controller outputs values in [-1, 1] via tanh;
          map throttle and brake to [0, 1]
        - Remember to preprocess observations (resize, normalize)
    """
    # TODO: Implement evaluation
    pass


def plot_comparison(baseline_rewards, wm_rewards):
    """
    Create a comparison plot of baseline vs. world model performance.

    Args:
        baseline_rewards: List of episode rewards for the baseline
        wm_rewards: List of episode rewards for the world model agent

    Tasks:
        1. Create a bar chart comparing mean rewards with error bars
        2. Create a box plot showing the distribution of rewards
        3. Print a summary table with mean, std, min, max for each
```

```
    Hints:
        - Use matplotlib subplots with 1 row, 2 columns
        - Error bars should show standard deviation
        - Include a horizontal line at reward=900 labeled
          "Human-level reference"
    """
    # TODO: Implement comparison visualization
    pass
```

Thought questions: - How does the agent's performance in the real environment compare to its performance in dreams? What explains the gap? - If the world model had perfect fidelity, would the dream performance and real performance be identical? Why or why not? - What is the sample efficiency ratio? How many real environment steps did the world model agent use compared to the model-free baseline's 10M steps?

3.7 Error Analysis

Context: Understanding failure modes is critical for FleetPath's safety case. A robot that performs well on average but catastrophically fails in specific scenarios is not deployable.

TODO 7 — Analyze failure modes:

```
def failure_analysis(env, vae, mdnrnn, controller,
                     n_episodes=20, max_steps=1000):
    """
    Identify and categorize failure modes of the trained agent.

    Args:
        env: Gymnasium environment
        vae, mdnrnn, controller: Trained model components
        n_episodes: Number of episodes to analyze
        max_steps: Maximum steps per episode

    Returns:
        failures: List of dicts, each containing:
            - episode: Episode number
            - step: Step where failure occurred
            - type: Category of failure (see below)
            - observation: Frame at failure point
            - latent: Latent code at failure point
            - dream_prediction: What the world model predicted
            - actual_outcome: What actually happened

    Failure categories to detect:
        1. "off_track" — car leaves the road
        2. "stall" — car speed drops to near zero
        3. "oscillation" — rapid left-right steering
        4. "model_exploitation" — agent finds a "cheat" in the
           dream that does not transfer to reality

    Steps:
        1. Run episodes and record frame-by-frame data
        2. Detect failures using heuristics:
           - off_track: reward < -1 for consecutive steps
           - stall: speed < 0.01 for > 10 steps
           - oscillation: steering changes sign > 5 times in
             10 steps
           - model_exploitation: dream reward >> real reward
        3. For each failure, save the context (5 frames before
           and after)
        4. Summarize: count failures by category, compute failure
           rate

    Hints:
        - Track the running reward to detect off-track events
        - Compare dream-predicted z_{t+1} with actual encoded
          z_{t+1} to detect model exploitation
```

```
    """
    # TODO: Implement failure analysis
    pass
```

Thought questions: - Which failure modes are most concerning for warehouse deployment? Why? - How could the world model be improved to reduce model exploitation failures? - If you were FleetPath's safety engineer, what minimum failure rate would you require before approving deployment?

3.8 Scalability and Deployment Considerations

Context: FleetPath needs to deploy this model on NVIDIA Jetson AGX Orin with < 100ms inference latency. The student profiles the inference pipeline.

TODO 8 — Profile inference performance:

```
def profile_inference(vae, mdnrnn, controller, device='cpu'):
    """
    Profile the inference latency of each component.

    Args:
        vae: Trained VAE
        mdnrnn: Trained MDN-RNN
        controller: Trained Controller
        device: Device to profile on ('cpu' or 'cuda')

    Returns:
        timing: Dictionary with per-component latency in ms

    Steps:
        1. Create a dummy input (random 64x64x3 image)
        2. Time 100 forward passes through each component:
           a. VAE encoder: x -> z
           b. MDN-RNN: (z, a) -> (pi, mu, sigma, h)
           c. Controller: (z, h) -> a
           d. Full pipeline: x -> z -> (z, h) -> a
        3. Report mean and std of latency for each component
        4. Report total pipeline latency
        5. Determine if the 100ms (10 Hz) budget is met

    Hints:
        - Use torch.cuda.synchronize() before timing on GPU
        - Warm up with 10 untimed passes before measuring
        - Use time.perf_counter() for precise timing
    """
    # TODO: Implement inference profiling
    pass
```

Thought questions: - Which component is the bottleneck? Why? - How would you reduce latency if the 100ms budget is not met? - What is the trade-off between model size and dream fidelity?

3.9 Ethical and Regulatory Analysis

Context: FleetPath's robots operate alongside human workers. Safety, fairness, and regulatory compliance are not optional.

TODO 9 — Ethical impact assessment:

```
def ethical_assessment():
    """
```

```
    Write a brief ethical impact assessment for deploying a
    world-model-based navigation system in a fulfillment center.

    Address the following questions (write your answers as a
    multi-line string and print them):

    1. SAFETY: The world model may have learned inaccurate
       dynamics in certain regions of the state space. How should
       the system detect and respond to situations where the
       world model's predictions diverge from reality? What
       fallback mechanisms should be in place?

    2. WORKER IMPACT: Autonomous robots change the nature of
       warehouse work. What safeguards should be in place to
       ensure worker safety? How should the system handle
       edge cases where a human worker is in an unexpected
       position?

    3. BIAS: If the training data was collected primarily during
       day shifts, the world model may perform poorly during
       night shifts (different lighting, fewer workers, different
       traffic patterns). How should this be detected and
       mitigated?

    4. ACCOUNTABILITY: If a robot trained in dreams causes an
       injury in the real world, who is responsible? The ML
       engineer who trained the world model? The CMA-ES
       optimizer? The company? How should accountability be
       structured?

    5. REGULATORY: What standards or certifications would
       FleetPath need before deploying dream-trained robots?
       Consider ISO 3691-4 (industrial truck safety), OSHA
       warehouse safety standards, and emerging AI regulation.

    Return your answers as a formatted string.
    """
    # TODO: Write your ethical assessment
    assessment = """
    [Write your assessment here]
    """
    print(assessment)
    return assessment
```

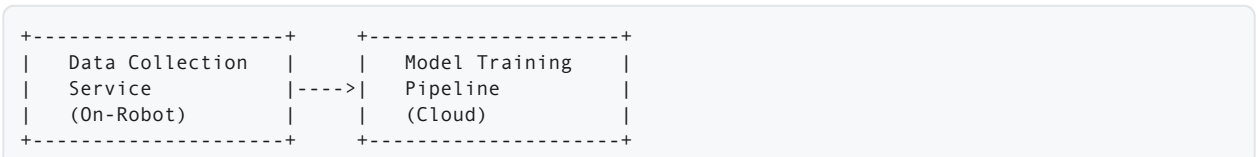
Thought questions: - Should dream-trained robots be held to a higher safety standard than conventionally trained robots? Why or why not? - How does the "model exploitation" problem interact with safety requirements? - If you discovered that the world model consistently underestimates the speed of human workers, what would you do?

Section 4: Production and System Design Extension

This section describes how FleetPath would deploy the world model navigation system at production scale. It is intended for advanced students interested in ML systems engineering.

Architecture Overview

The production system has four major subsystems:



Data Collection Service: Runs on each robot. Records camera frames, actions, and rewards during normal operation. Periodically uploads batches to the cloud training pipeline. Implements a circular buffer retaining the most recent 48 hours of data per robot.

Model Training Pipeline: Cloud-based training orchestration. Retrains the VAE on new visual data, updates the MDN-RNN with new dynamics, and evolves the controller in dreams. Triggered weekly or when drift detection exceeds thresholds.

Model Registry: Stores versioned snapshots of all three components (V, M, C) with metadata including training data hash, evaluation metrics, and approval status. Supports instant rollback to any previous version.

Real-Time Inference: On-robot inference running on NVIDIA Jetson AGX Orin. Executes the full V-M-C pipeline at 10 Hz. Falls back to a conservative rule-based controller if the world model's prediction confidence drops below a threshold.

Monitoring and Drift Detection: Cloud-based monitoring of per-robot performance metrics, world model prediction accuracy, and environment change detection.

API Design

Inference API (gRPC, on-robot):

```
service NavigationService {
    rpc GetAction(ObservationRequest) returns (ActionResponse);
    rpc GetWorldModelState(StateRequest) returns (WorldModelState);
    rpc ResetEpisode(ResetRequest) returns (ResetResponse);
}

message ObservationRequest {
    bytes image_data = 1;       // 64x64x3 RGB, uint8
    float timestamp = 2;
    repeated float prev_action = 3;  // [steering, throttle, brake]
}

message ActionResponse {
    repeated float action = 1;  // [steering, throttle, brake]
    float confidence = 2;       // World model prediction confidence
    float latency_ms = 3;
}
```

Training API (REST, cloud):

```
POST /api/v1/training/trigger
    Body: { "warehouse_id": "WH-042", "data_window_hours": 48 }
    Response: { "job_id": "train-20250215-001", "status": "queued" }
```

```
GET /api/v1/models/{version}
    Response: { "version": "v2.3.1", "metrics": {...}, "status": "approved" }

POST /api/v1/models/{version}/deploy
    Body: { "robot_ids": ["R-001", "R-002"], "rollout_percent": 10 }
    Response: { "deployment_id": "dep-001", "status": "rolling_out" }
```

Serving Infrastructure

- **On-Robot**: NVIDIA Jetson AGX Orin (32GB). Models loaded as TorchScript (.pt) for optimized inference. The VAE encoder, LSTM forward pass, and linear controller execute sequentially. Total model size: ~30MB.
- **Cloud Training**: AWS p3.2xlarge (1x V100) for VAE and MDN-RNN training. Controller CMA-ES runs on CPU instances (c5.4xlarge) since dream rollouts are compute-light. Training pipeline orchestrated by AWS Step Functions.
- **Scaling**: Each robot runs inference independently. Cloud training scales horizontally — one training job per warehouse, parallelized dream rollouts across CPU cores.

Latency Budget

Component	Latency (ms)	Notes
Image preprocessing (resize, normalize)	2	CPU, optimized with NumPy
VAE encoder forward pass	12	Jetson GPU, TorchScript
LSTM forward pass	5	Jetson GPU
MDN head computation	1	Jetson GPU
Controller forward pass	< 1	CPU-feasible (linear layer)
Action post-processing	1	Clipping, safety bounds
Total	**~22 ms**	**Well within 100ms budget**
Safety margin	78 ms	Available for safety checks, logging

Monitoring

Key metrics tracked per robot:

Metric	Alert Threshold	Dashboard
Prediction error (L2 between predicted and actual z)	> 2x rolling average	Per-robot time series
Episode reward (rolling 100-episode average)	< 700	Fleet-wide heatmap
Collision rate	> 0 in production	Instant alert + incident report
Inference latency (p99)	> 80ms	Latency histogram
World model confidence (min pi value)	< 0.1	Distribution plot

Metric	Alert Threshold	Dashboard
Near-miss events (distance to human < 0.5m)	> 0	Instant alert

Dashboards: Grafana dashboards showing fleet-wide metrics aggregated by warehouse, shift, and robot. Separate dashboards for ML model health (prediction accuracy, latent space drift) and operational health (uptime, throughput, collision events).

Model Drift Detection

The world model can become stale as the warehouse environment changes (new shelf layouts, different product sizes, seasonal worker count changes). FleetPath detects drift through three mechanisms:

1. **Prediction residual monitoring**: Track the L2 distance between the MDN-RNN's predicted next latent state and the actual encoded next observation. A sustained increase (> 2 standard deviations above the 30-day rolling mean for > 4 hours) triggers a retraining alert.
2. **Latent distribution shift**: Compare the distribution of encoded latent codes z_t against a reference distribution from the training data using the Maximum Mean Discrepancy (MMD) statistic. An MMD value exceeding a threshold triggers investigation.
3. **Performance degradation**: If the rolling 100-episode average reward drops below 700 (from a target of 850+), the system flags potential drift and automatically switches affected robots to the conservative fallback controller while retraining begins.

Model Versioning

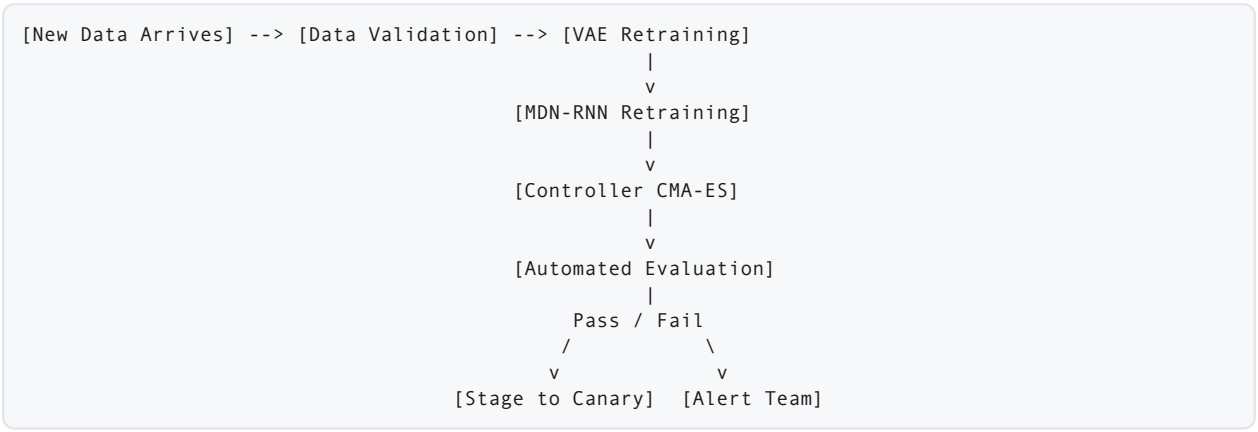
- **Versioning scheme**: Semantic versioning (e.g., v2.3.1) where major version = architecture change, minor version = retraining on new data, patch version = hyperparameter tuning.
- **Storage**: All model artifacts (VAE, MDN-RNN, Controller weights + training config + evaluation results) stored in S3 with DynamoDB metadata.
- **Rollback**: One-command rollback to any previous version. The model registry maintains a "last known good" pointer that is automatically updated only after a model passes all validation gates.

A/B Testing

- **Canary deployment**: New models are first deployed to 10% of robots in a single warehouse for 48 hours.
- **Guardrail metrics**: Zero collisions, reward > 800, latency p99 < 80ms. If any guardrail is violated, automatic rollback triggers within 60 seconds.
- **Statistical significance**: Use a two-sample t-test on episode rewards with $\alpha = 0.05$ and minimum power of 0.8. This requires approximately 50 episodes per group, achievable in ~8 hours of operation.

- **Progressive rollout**: After passing canary, expand to 25% -> 50% -> 100% of robots in the warehouse, then to other warehouses. Each stage requires 24 hours of clean operation.

CI/CD for ML



Validation gates at each stage:

1. **Data validation**: Check for corrupted frames (> 5% black/white pixels), action range violations, minimum episode length (> 50 steps)
2. **VAE validation**: Reconstruction loss < 1.5x previous version, KL divergence within expected range, visual inspection of 20 random reconstructions
3. **MDN-RNN validation**: Prediction log-likelihood > 90% of previous version, 100-step dream rollouts produce visually plausible sequences
4. **Controller validation**: Average dream reward > 800, average real reward > 800 (evaluated on 50 real episodes in a test zone), zero collisions in test episodes
5. **Integration test**: Full pipeline inference latency < 80ms on Jetson hardware

Cost Analysis

Training costs (per warehouse onboarding):

Item	Cost
Data collection (2 days teleoperation labor)	USD 2,400
VAE training (4 hours on p3.2xlarge)	USD 50
MDN-RNN training (6 hours on p3.2xlarge)	USD 75
Controller CMA-ES (8 hours on c5.4xlarge x 4)	USD 22
Evaluation rollouts (2 hours real-world)	USD 200
Total per warehouse	**USD 2,747**

Comparison with current approach:

	Model-Free (Current)	World Model (Proposed)
Onboarding time	6-8 weeks	3-5 days

	Model-Free (Current)	World Model (Proposed)
Onboarding cost	USD 928,000	USD 2,747
Training collisions	~340	0
Real environment steps	8-12M	~50,000
Compute cost	USD 500 (GPU time)	USD 147 (GPU time)

Annual inference costs (per robot):

Item	Cost
Jetson AGX Orin (amortized over 3 years)	USD 500/year
Power consumption (15W continuous)	USD 13/year
Cloud monitoring and logging	USD 50/year
Total per robot per year	**USD 563**

For FleetPath's fleet of 200 robots across 8 warehouses, the annual inference cost is approximately USD 112,600 — a fraction of the USD 13M projected onboarding cost under the current model-free approach.

This case study was developed for the Vizuara AI curriculum. The company profile, financial figures, and deployment details are illustrative and based on publicly available industry benchmarks.