Case Study: Autonomous Pick-and-Place for High-SKU E-Commerce Fulfillment

A World Model + Vision-Language-Action Approach to Generalizable Robotic Manipulation

Section 1: Industry Context and Business Problem

Industry: E-Commerce Fulfillment and Warehouse Robotics

The global warehouse automation market is valued at approximately $23 billion (2025) and is projected to exceed $40 billion by 2030. E-commerce order volumes have tripled since 2019, but the labor pool for warehouse work is shrinking -- turnover rates in U.S. fulfillment centers exceed 100% annually. The economics are brutal: the average cost to pick, pack, and ship a single order is $5.50, with labor accounting for 65% of that cost.

Robotic pick-and-place is the critical bottleneck. While mobile robots (AMRs) have solved the problem of moving bins through warehouses, the act of reaching into a bin, identifying the correct item, grasping it without damage, and placing it into a shipping box remains largely unsolved for the general case.

Company Profile: NovaPick Robotics

- **Founded:** 2021, San Jose, CA
- **Team:** 118 employees (42 engineers, 12 ML researchers, 8 robotics engineers)
- **Funding:** Series B, $45M raised ($8M seed from Lux Capital, $37M Series A+B led by Andreessen Horowitz)
- **Revenue:** $6.2M ARR from 4 fulfillment center deployments
- **Product:** NovaPick Cell -- a modular robotic pick-and-place station consisting of a 6-DOF Franka Emika Panda arm, a wrist-mounted Intel RealSense D435 depth camera, a suction/parallel-jaw hybrid gripper, and proprietary control software
- **Deployment:** Currently operational in fulfillment centers for a mid-size online retailer (StyleBox), a specialty food distributor (FreshRoute), and two 3PL warehouses

Business Challenge

NovaPick's current system uses a classical pipeline: point cloud segmentation, object pose estimation, analytical grasp planning (GraspIt!), and trajectory optimization (TrajOpt). This pipeline works reliably for a curated catalog of ~500 SKUs that have been individually scanned, modeled, and tested.

The problem is scale.

NovaPick's largest prospective client, **MegaMart** (a top-5 U.S. online retailer), operates 14 fulfillment centers processing 2.1 million orders per day across **83,000 active SKUs**. The SKU catalog changes by approximately 15% per quarter as products are added, discontinued, or repackaged.

Under the current engineering-intensive approach: - Each new SKU requires 2-4 hours of engineering time for 3D scanning, grasp annotation, and validation - At $150/hour fully loaded engineer cost, onboarding 83,000 SKUs would cost **$24.9M -- $49.8M** - Seasonal SKU turnover (~12,450 new SKUs per quarter) would require a permanent team of 15-20 engineers doing nothing but SKU onboarding - Even then, the system fails on deformable objects (bags, pouches, clothing), transparent items (bottles, blister packs), and items with unusual geometries

MegaMart has offered NovaPick a **$180M, 5-year contract** to automate pick-and-place across all 14 fulfillment centers -- but only if NovaPick can demonstrate **zero-shot generalization** to novel objects with a pick success rate above 90%.

Why It Matters

- **Revenue at stake:** $180M contract, plus an estimated $500M in follow-on contracts from other retailers watching this deployment
- **Unit economics:** MegaMart currently spends $1.2B/year on warehouse labor. Even a 30% reduction through automation saves $360M/year
- **Competitive pressure:** Covariant (acquired by Amazon), Dexterity AI, and RightHand Robotics are all pursuing the same contract
- **Timeline:** MegaMart requires a proof-of-concept demo within 6 months, with full deployment in 18 months

Constraints

Constraint	Requirement
Pick cycle time	< 4 seconds per item (15 picks/minute)
Pick success rate	> 90% on novel objects, > 97% on known objects
Compute budget (inference)	Single NVIDIA L4 GPU per cell (24GB VRAM, ~$0.80/hr cloud equivalent)
Compute budget (training)	128 A100 GPUs for up to 2 weeks (allocated from NovaPick's existing cloud contract)
Latency	< 200ms from image capture to first joint command
Safety	ISO 10218-1 compliant; must stop within 50ms if anomaly detected

Constraint	Requirement
Data	No MegaMart proprietary data available pre-contract; must train on public + synthetic data
Gripper	Suction + parallel-jaw hybrid; model must output gripper mode selection
Environment	Semi-structured bins with 1-30 items; variable lighting; items may be partially occluded

Section 2: Technical Problem Formulation

Problem Type

This is a **conditional continuous action generation** problem. Given a visual observation of a bin and a language instruction specifying the target item, the system must generate a continuous trajectory of robot joint angles and gripper commands that successfully picks the specified item and places it at a target location.

The problem can be decomposed into two coupled sub-problems:

1. **World modeling:** Learn a latent dynamics model that predicts the outcome of candidate grasp actions (will the grasp succeed? will adjacent items be displaced?) without physical execution
2. **Action generation:** Generate smooth, collision-free robot trajectories conditioned on visual input, language specification, and world model predictions

Input Specification

Input	Format	Dimensions
RGB image (wrist camera)	Float32 tensor	(3, 480, 640)
Depth map (wrist camera)	Float32 tensor	(1, 480, 640)
Language instruction	Tokenized text	Variable length, max 64 tokens
Robot proprioceptive state	Float32 vector	(14,) -- 7 joint angles + 7 joint velocities
Gripper state	Float32 vector	(2,) -- suction pressure + jaw aperture

Output Specification

Output	Format	Dimensions	Frequency
Joint velocity commands	Float32 vector	(7,)	20 Hz
Gripper mode	Categorical	{suction, parallel-jaw, release}	Per action chunk
Grasp confidence	Float32 scalar	(1,)	Per action chunk

Output	Format	Dimensions	Frequency
Action chunk	Float32 tensor	(16, 10)	Every 0.8s

The model outputs **action chunks** of 16 timesteps (0.8 seconds at 20 Hz), following the chunked action generation paradigm from pi-0. Each chunk is a smooth trajectory segment; chunks are blended together for continuous execution.

Loss Function

The training loss is a composite of three terms:

1. Flow matching loss (action generation):

$$\mathcal{L}_{\text{flow}} = \mathbb{E}_{t,\epsilon}\left[\|v_\theta(a_t, t, o, l) - (a_1 - a_0)\|^2\right]$$

where $a_t = (1-t)\cdot a_0 + t\cdot a_1$ is the interpolation between noise a_0 and ground-truth action a_1 , o is the visual observation, l is the language embedding, and v_θ is the learned velocity field.

2. World model prediction loss (RSSM-style):

$$\mathcal{L}_{\text{wm}} = \mathbb{E}\left[\|\hat{z}_{t+1} - z_{t+1}\|^2 + \beta\cdot D_{KL}(q(z_t|h_t, o_t)\|p(z_t|h_t))\right]$$

where $\hat{z}_{t+1}$ is the predicted next latent state, z_{t+1} is the encoded actual next state, and the KL term encourages the prior (imagination) distribution to match the posterior (observation-informed) distribution.

3. Grasp success prediction loss:

$$\mathcal{L}_{\text{grasp}} = -\left[y\log(\hat{y}) + (1-y)\log(1-\hat{y})\right]$$

where $y \in \{0,1\}$ is the ground-truth grasp outcome and $\hat{y}$ is the predicted success probability.

Total loss:

$$\mathcal{L} = \mathcal{L}_{\text{flow}} + \lambda_1\mathcal{L}_{\text{wm}} + \lambda_2\mathcal{L}_{\text{grasp}}$$

with $\lambda_1 = 0.5$ and $\lambda_2 = 0.1$ (tuned on validation set).

Justification: Flow matching is chosen over diffusion for action generation because it provides straighter transport paths and faster inference (fewer denoising steps). The RSSM-style world model loss follows DreamerV3's formulation, enabling imagination-based trajectory evaluation. Binary cross-entropy for grasp prediction provides a direct signal for the most business-critical outcome.

Evaluation Metrics

Metric	Definition	Target
Pick success rate (primary)	Fraction of attempts where the target item is successfully grasped and lifted	> 90% (novel), > 97% (known)
Cycle time	Time from image capture to item placed in box	< 4.0 seconds
Grasp damage rate	Fraction of picks causing item damage	< 0.5%
Action smoothness	Mean jerk (3rd derivative of position) across trajectory	< 500 rad/s^3
World model prediction accuracy	Cosine similarity between predicted and actual next latent state	> 0.85
Inference latency	Time from observation to first action chunk	< 200ms
Collision rate	Fraction of trajectories with unplanned contact	< 1%

Baseline

Classical pipeline (current NovaPick system): - Point cloud segmentation (Mask R-CNN) + 6-DOF pose estimation (DenseFusion) + analytical grasp planning (GraspIt!) + trajectory optimization (TrajOpt) - Performance on known SKUs: 96.2% pick success, 3.1s cycle time - Performance on novel SKUs: 34.7% pick success (catastrophic failure) - Requires per-SKU 3D model registration

Simple learned baseline: - ResNet-50 backbone + MLP grasp quality predictor (similar to Dex-Net 4.0) - Performance on novel SKUs: ~72% pick success (better generalization but no temporal reasoning, no language conditioning, jerky motions)

Why World Models + VLA

The article's core concepts -- world models and Vision-Language-Action architectures -- are the right technical approach for this problem because:

1. **World models enable mental rehearsal:** Before executing a grasp, the system can imagine the outcome in latent space. Will the suction cup seal on this surface? Will pulling this item cause adjacent items to tumble? This reduces physical trial-and-error, which is expensive and dangerous in a real warehouse.

2. **VLA architecture enables zero-shot generalization:** By training on diverse manipulation data with language grounding, the model learns a general mapping from (vision, language) to action. A new SKU described as "small red plastic bottle" can be grasped without any SKU-specific engineering because the model understands both the visual appearance and the physical affordances implied by the description.

3. **Flow matching enables smooth trajectories:** Unlike discrete grasp pose prediction (which requires a separate motion planner), flow-matching VLA models generate smooth,

continuous action trajectories end-to-end. This eliminates the classical pipeline's handoff between grasp planning and motion planning.

4. **Abstract prediction avoids pixel-level waste:** Following the JEPA insight from the article, predicting grasp outcomes in latent space rather than pixel space allows the world model to focus on what matters (object pose, stability, contact forces) rather than irrelevant details (texture, exact lighting).

Technical Constraints

Constraint	Value
Model parameter budget	< 1B parameters (must fit on L4 GPU with action chunk batch)
Inference compute	< 200ms per action chunk on NVIDIA L4
Training compute	128x A100-80GB for up to 14 days
Training data	~500K grasp episodes from public datasets + ~200K synthetic episodes
Action chunk size	16 timesteps at 20 Hz = 0.8s lookahead
World model imagination horizon	5 steps (4 seconds lookahead)

Section 3: Implementation Notebook Structure

This section outlines a Google Colab notebook that guides students through building a simplified version of the NovaPick system. The notebook uses publicly available datasets and a simulated environment to keep compute requirements manageable on a free Colab GPU.

3.1 Data Acquisition Strategy

Dataset: The primary dataset is a combination of: - **DROID (Distributed Robot Interaction Dataset):** 76K real-world robot manipulation trajectories across diverse environments and objects, with RGB-D images, proprioceptive states, and action labels - **Google Robot dataset (from Open X-Embodiment):** 53K pick-and-place episodes with language annotations - **Synthetic augmentation:** PyBullet-generated grasp episodes with procedurally generated objects

For the notebook, students will work with a curated subset of 10K episodes from DROID and 5K from Google Robot, pre-processed and hosted on Hugging Face.

Preprocessing pipeline: - Resize RGB images to (3, 224, 224) - Normalize depth maps to [0, 1] range - Tokenize language instructions using a frozen CLIP text encoder - Normalize joint angles to [-1, 1] range - Segment episodes into overlapping action chunks of 16 timesteps

TODO: Data Augmentation Pipeline

```
def augment_observation(rgb: torch.Tensor, depth: torch.Tensor,
                         language: str) -> tuple[torch.Tensor, torch.Tensor, str]:
    """
    Apply data augmentation to a single observation.

    Args:
        rgb: RGB image tensor of shape (3, 224, 224), values in [0, 1]
        depth: Depth map tensor of shape (1, 224, 224), values in [0, 1]
        language: Language instruction string

    Returns:
        Augmented (rgb, depth, language) tuple

    Requirements:
        1. Apply random color jitter to RGB (brightness +/-0.2, contrast +/-0.2,
           saturation +/-0.2) -- this simulates variable warehouse lighting
        2. Apply the SAME random crop (scale 0.8-1.0) to BOTH rgb and depth --
           spatial correspondence must be preserved
        3. Apply random horizontal flip with p=0.5 to BOTH rgb and depth --
           remember to also flip any spatial terms in the language instruction
           (swap "left" <-> "right")
        4. Add Gaussian noise (std=0.01) to the depth map to simulate sensor noise

    Hints:
        - Use torchvision.transforms for color jitter
        - For paired spatial transforms, generate random parameters first, then
          apply the same params to both modalities
        - For language flipping, use simple string replacement
    """
    # YOUR CODE HERE
    raise NotImplementedError
```

Verification cell:

```
# Test your augmentation
rgb_test = torch.rand(3, 224, 224)
depth_test = torch.rand(1, 224, 224)
lang_test = "pick up the red bottle on the left"

rgb_aug, depth_aug, lang_aug = augment_observation(rgb_test, depth_test, lang_test)

assert rgb_aug.shape == (3, 224, 224), f"RGB shape mismatch: {rgb_aug.shape}"
assert depth_aug.shape == (1, 224, 224), f"Depth shape mismatch: {depth_aug.shape}"
assert rgb_aug.min() >= 0.0, "RGB values must be non-negative"
assert isinstance(lang_aug, str), "Language must be a string"
print("Augmentation tests passed.")
```

3.2 Exploratory Data Analysis

Students explore the dataset to understand its structure, distributions, and potential challenges.

Key distributions to plot: - Distribution of action magnitudes per joint (are some joints used more than others?) - Distribution of episode lengths (are most episodes short or long?) - Language instruction length distribution and word frequency analysis - Depth map quality histogram (what fraction of pixels have valid depth?) - Gripper mode distribution across the dataset (suction vs. parallel-jaw vs. release)

Anomalies to investigate: - Episodes with near-zero action variance (robot not moving -- possible data collection errors) - Depth maps with > 30% invalid pixels (sensor occlusion or failure) - Mismatched language instructions (instruction mentions object not visible in frame)

TODO: EDA Implementation

```
def compute_dataset_statistics(dataloader: DataLoader) -> dict:
    """
    Compute comprehensive statistics over the entire dataset.

    Args:
        dataloader: PyTorch DataLoader yielding (rgb, depth, language, actions,
                    proprioception) tuples

    Returns:
        Dictionary with keys:
            - "action_mean": mean action per joint, shape (7,)
            - "action_std": std action per joint, shape (7,)
            - "episode_lengths": list of episode lengths
            - "depth_valid_fraction": list of valid pixel fractions per sample
            - "num_anomalous_episodes": count of episodes with action std < 0.01

    Hints:
        - Use running statistics (Welford's algorithm) to avoid loading all
          data into memory
        - For depth validity, count pixels where depth > 0.01 (near-zero
          depth indicates invalid measurement)

    Thought questions (answer in a markdown cell after your code):
        1. Which joints have the highest action variance? What does this tell
           you about the most common manipulation motions?
        2. What fraction of episodes are "anomalous" (near-zero action)? Should
           these be filtered or kept? What are the tradeoffs?
        3. Is the depth data quality sufficient for learning, or do you need
           additional preprocessing?
    """
    # YOUR CODE HERE
    raise NotImplementedError
```

3.3 Baseline Approach

Before building the world model + VLA system, students implement a simple baseline to establish a performance floor.

Baseline: Image-conditioned grasp quality network

Architecture: frozen ResNet-18 visual encoder + 3-layer MLP that predicts (x, y, z, roll, pitch, yaw, gripper_width) for a single grasp pose. No temporal reasoning, no language conditioning, no world model.

TODO: Implement Baseline Model

```
class GraspBaseline(nn.Module):
    """
    Simple grasp prediction baseline.

    Takes an RGB-D image and predicts a single grasp pose.
    No language input, no temporal reasoning, no world model.

    Architecture:
        1. Frozen ResNet-18 backbone (pretrained on ImageNet) processes the
           RGB image -> 512-dim feature vector
        2. A small CNN (3 conv layers) processes the depth map -> 128-dim
           feature vector
        3. Concatenate visual features (512 + 128 = 640 dims)
        4. MLP: 640 -> 256 -> 128 -> 7 (grasp pose)
        5. Output: (x, y, z, roll, pitch, yaw, gripper_width)

    Hints:
        - Use torchvision.models.resnet18(pretrained=True) and freeze with
          requires_grad_(False)
        - Remove the final FC layer of ResNet; use the 512-dim avgpool output
        - The depth CNN should use (1, 224, 224) input with channels:
          1 -> 32 -> 64 -> 128, kernel size 3, stride 2, with batch norm
        - Apply tanh to the output to constrain predictions to [-1, 1]
    """
    def __init__(self):
        super().__init__()
        # YOUR CODE HERE
        raise NotImplementedError

    def forward(self, rgb: torch.Tensor, depth: torch.Tensor) -> torch.Tensor:
        """
        Args:
            rgb: (B, 3, 224, 224) RGB image
            depth: (B, 1, 224, 224) depth map
        Returns:
            grasp_pose: (B, 7) predicted grasp pose
        """
        # YOUR CODE HERE
        raise NotImplementedError
```

Evaluation: Students train this baseline on the dataset using MSE loss against ground-truth grasp poses and evaluate pick success rate in a PyBullet simulation environment provided in the notebook.

3.4 Model Design

This is the core of the case study. Students implement a simplified world model + VLA system inspired by the concepts from the article.

Architecture overview:

The system has three main components:

1. **Observation Encoder:** A multimodal encoder that fuses RGB, depth, and language into a unified representation
2. **World Model (RSSM):** A recurrent state-space model that predicts future latent states given actions
3. **Action Generator (Flow Matching):** A conditional flow model that generates action chunks from noise, conditioned on the observation encoding and world model state

TODO: Implement Observation Encoder

```
class ObservationEncoder(nn.Module):
    """
    Multimodal encoder that fuses vision (RGB + depth) and language into a
    single latent representation.

    Architecture:
        1. Visual encoder: Use a frozen DINOv2 ViT-S/14 backbone to encode
           the RGB image into patch tokens. Concatenate depth as a 4th channel
           via a learned linear projection.
           Output: (B, N_patches, 384)

        2. Language encoder: Use a frozen CLIP text encoder to get a language
           embedding.
           Output: (B, 512)

        3. Cross-attention fusion: Use 2 layers of cross-attention where
           language tokens attend to visual patch tokens. This allows the model
           to focus on the part of the image relevant to the instruction.
           Output: (B, 256)
        4. Final projection: Linear layer to produce the observation embedding.
           Output: (B, 256)

    Hints:
        - For DINOv2, use torch.hub.load('facebookresearch/dinov2', 'dinov2_vits14')
        - For CLIP, use clip.load("ViT-B/32") and freeze
        - In cross-attention, language is the query, visual patches are key/value
        - Use nn.MultiheadAttention with 4 heads and embed_dim=384
        - Add a learnable [CLS] token to the language side for pooling

    Thought question:
        Why is cross-attention better than simple concatenation for fusing
        vision and language? Think about what happens when the instruction says
        "pick up the RED bottle" -- which image patches should receive the
        most attention?
    """
    def __init__(self, obs_embed_dim: int = 256):
        super().__init__()
        # YOUR CODE HERE
        raise NotImplementedError

    def forward(self, rgb: torch.Tensor, depth: torch.Tensor,
                language_tokens: torch.Tensor) -> torch.Tensor:
        """
        Args:
            rgb: (B, 3, 224, 224)
            depth: (B, 1, 224, 224)
            language_tokens: (B, 77) CLIP tokenized text
        Returns:
            obs_embed: (B, 256) fused observation embedding
        """
        # YOUR CODE HERE
        raise NotImplementedError
```

TODO: Implement RSSM World Model

```
class RSSM(nn.Module):
    """
    Recurrent State-Space Model for world modeling.

    This is the core world model from DreamerV3, adapted for manipulation.
    It maintains both a deterministic recurrent state and a stochastic
    latent state.

    State components:
        - h_t: deterministic state (GRU hidden state), shape (B, 512)
        - z_t: stochastic state (categorical), shape (B, 32, 32)
          -- 32 categorical variables, each with 32 classes
          (flattened to (B, 1024) when needed)

    Sub-modules:
        1. Deterministic transition: GRU that takes [z_{t-1}, a_{t-1}] as
           input and h_{t-1} as hidden state -> h_t

        2. Prior (imagination): MLP that predicts z_t distribution from h_t
           alone (used during dreaming)
           h_t -> MLP(512, 512, 32*32) -> reshape to (B, 32, 32) ->
           softmax over last dim

        3. Posterior (encoder): MLP that predicts z_t distribution from
           [h_t, obs_embed] (used during training with real observations)
           [h_t, obs_embed] -> MLP(768, 512, 32*32) -> reshape -> softmax

        4. Reward predictor: MLP that predicts expected reward from [h_t, z_t]
           [h_t, z_t_flat] -> MLP(1536, 256, 1) -> output scalar

    Hints:
        - Use nn.GRUCell for the deterministic transition
        - For the stochastic state, use straight-through gradients with
          Gumbel-Softmax for differentiable sampling
        - The KL loss between prior and posterior encourages the prior to
          be useful for imagination
        - Start with temperature=1.0 for Gumbel-Softmax
    Thought question:
        Why does the RSSM use BOTH deterministic and stochastic states?
        Consider what would happen with only deterministic (cannot model
        uncertainty) or only stochastic (loses long-term memory) states.
    """
    def __init__(self, obs_embed_dim: int = 256, action_dim: int = 10,
                 det_dim: int = 512, stoch_dim: int = 32,
                 stoch_classes: int = 32):
        super().__init__()
        self.det_dim = det_dim
        self.stoch_dim = stoch_dim
        self.stoch_classes = stoch_classes
        # YOUR CODE HERE
        raise NotImplementedError

    def initial_state(self, batch_size: int, device: torch.device) -> tuple:
        """Return initial (h_0, z_0) with zeros."""
        # YOUR CODE HERE
        raise NotImplementedError

    def observe_step(self, obs_embed: torch.Tensor, action: torch.Tensor,
                     h_prev: torch.Tensor, z_prev: torch.Tensor
                     ) -> tuple[torch.Tensor, torch.Tensor, dict]:
        """
        Single step with real observation (training time).

        Returns: (h_t, z_t, info_dict)
            info_dict contains 'prior_logits', 'posterior_logits' for KL loss
        """
        # YOUR CODE HERE
        raise NotImplementedError

    def imagine_step(self, action: torch.Tensor, h_prev: torch.Tensor,
                     z_prev: torch.Tensor
                     ) -> tuple[torch.Tensor, torch.Tensor]:
        """
        Single step WITHOUT observation (imagination/dreaming).

        Returns: (h_t, z_t) using the prior distribution
        """
        # YOUR CODE HERE
        raise NotImplementedError

    def imagine_trajectory(self, initial_h: torch.Tensor,
                           initial_z: torch.Tensor,
                           actions: torch.Tensor) -> dict:
        """
        Roll out imagination for multiple steps.

        Args:
            initial_h: (B, 512) starting deterministic state
            initial_z: (B, 32, 32) starting stochastic state
            actions: (B, T, 10) sequence of actions to imagine

        Returns:
            dict with 'h_states': (B, T, 512), 'z_states': (B, T, 32, 32),
            'rewards': (B, T)
        """
        # YOUR CODE HERE
        raise NotImplementedError
```

TODO: Implement Flow Matching Action Generator

```
class FlowMatchingActionHead(nn.Module):
    """
    Generates action chunks via conditional flow matching.

    Instead of predicting actions directly (which produces jerky motions),
    this module learns a velocity field that transforms Gaussian noise into
    smooth action trajectories. This is the same principle used in pi-0.

    Architecture:
        - Input: noisy action chunk a_t, timestep t, conditioning c
```

```
          where c = [h, z_flat, obs_embed] from the world model
        - Network: 1D temporal U-Net (or simple MLP with timestep embedding)
          For simplicity, use an MLP variant:
          1. Embed timestep t using sinusoidal embedding -> (B, 128)
          2. Concatenate [a_t_flat, t_embed, c] -> (B, 16*10 + 128 + 1792)
          3. MLP: input_dim -> 1024 -> 512 -> 256 -> 16*10
          4. Reshape output to (B, 16, 10) -- velocity prediction

        - Training: Sample t ~ U(0,1), sample noise a_0 ~ N(0,I),
          interpolate a_t = (1-t)*a_0 + t*a_1 (where a_1 is ground truth),
          predict velocity v = a_1 - a_0

        - Inference: Start from a_0 ~ N(0,I), integrate velocity field
          using Euler method with N=10 steps

    Hints:
        - Sinusoidal embedding: [sin(t*freq), cos(t*freq)] for
          freq in geometric series
        - During training, the target velocity is simply (a_1 - a_0) --
          this is the optimal transport direction
        - Use 10 Euler integration steps at inference for good quality
          vs. speed tradeoff
        - Clip final actions to [-1, 1]

    Thought question:
        Why is flow matching preferred over a standard diffusion model
        (DDPM) for action generation? Think about: (a) the straightness
        of transport paths, (b) the number of inference steps needed,
        and (c) the smoothness of generated trajectories.
    """
    def __init__(self, action_chunk_size: int = 16, action_dim: int = 10,
                 cond_dim: int = 1792):
        super().__init__()
        self.action_chunk_size = action_chunk_size
        self.action_dim = action_dim
        # YOUR CODE HERE
        raise NotImplementedError

    def forward(self, noisy_actions: torch.Tensor, timestep: torch.Tensor,
                condition: torch.Tensor) -> torch.Tensor:
        """
        Predict velocity field.

        Args:
            noisy_actions: (B, 16, 10) noisy action chunk
            timestep: (B,) diffusion timestep in [0, 1]
            condition: (B, 1792) conditioning from world model
        Returns:
            velocity: (B, 16, 10) predicted velocity
        """
        # YOUR CODE HERE
        raise NotImplementedError

    def sample(self, condition: torch.Tensor, num_steps: int = 10
               ) -> torch.Tensor:
        """
        Generate action chunk from noise via Euler integration.

        Args:
            condition: (B, 1792) conditioning from world model
            num_steps: number of Euler integration steps
        Returns:
            actions: (B, 16, 10) generated action chunk
        """
        # YOUR CODE HERE
        raise NotImplementedError
```

3.5 Training Strategy

Optimizer: AdamW with weight decay 1e-4

Learning rate schedule: - Warmup: linear from 0 to 3e-4 over first 5K steps - Cosine decay from 3e-4 to 1e-5 over remaining training - Total: 200K gradient steps

Regularization: - Dropout 0.1 in MLPs - Gradient clipping: max norm 100 - KL balancing (DreamerV3 style): 80% free nats, clipped KL loss

Training loop structure: 1. Sample a batch of episodes from the dataset 2. Encode observations through the observation encoder 3. Run the RSSM forward with real observations (observe mode) to get latent states 4. Compute world model loss: reconstruction + KL 5. Sample random timesteps and noise for flow matching 6. Compute flow matching loss on action chunks 7. Backpropagate and update all parameters jointly

TODO: Implement Training Loop

```
def train_one_epoch(model: dict, dataloader: DataLoader,
                    optimizer: torch.optim.Optimizer,
                    scheduler: torch.optim.lr_scheduler._LRScheduler,
                    device: torch.device, epoch: int) -> dict:
    """
    Train all model components for one epoch.

    Args:
        model: dict with keys 'encoder', 'rssm', 'flow_head'
               (each an nn.Module)
        dataloader: yields batches of (rgb, depth, language, actions,
                    proprioception, rewards)
        optimizer: shared optimizer for all components
        scheduler: learning rate scheduler
        device: torch device
        epoch: current epoch number

    Returns:
        dict with 'flow_loss', 'wm_loss', 'kl_loss', 'total_loss'
        (averaged over epoch)

    Implementation steps:
        1. For each batch:
           a. Move all tensors to device
           b. Encode observations: obs_embed = encoder(rgb, depth, language)
           c. Initialize RSSM state: h, z = rssm.initial_state(B, device)
           d. Loop over timesteps in the episode:
              - h, z, info = rssm.observe_step(obs_embed[:, t], actions[:, t], h, z)
              - Accumulate KL loss from info['prior_logits'] vs info['posterior_logits']
              - Store h, z for action generation
           e. Compute world model loss (KL + reward prediction)
           f. Sample random t ~ U(0,1) for flow matching
           g. Create noisy actions: a_t = (1-t)*noise + t*ground_truth_actions
           h. Predict velocity: v_pred = flow_head(a_t, t, condition)
           i. Flow loss = MSE(v_pred, ground_truth_actions - noise)
           j. Total loss = flow_loss + 0.5 * wm_loss + 0.1 * grasp_loss
           k. Backprop and optimizer step

        2. Log losses every 100 steps using print or wandb
        3. Step the scheduler after each batch

    Hints:
        - Use torch.distributions.kl_divergence for KL computation between
          categorical distributions
        - Detach the RSSM hidden state at the start of each episode to prevent
          backprop through time explosion
        - Use gradient clipping: torch.nn.utils.clip_grad_norm_(params, 100)
    """
    # YOUR CODE HERE
    raise NotImplementedError
```

3.6 Evaluation

Quantitative evaluation: - Run the trained model in a PyBullet pick-and-place simulation (provided in the notebook) - Evaluate on two object sets: (a) 50 objects seen during training, (b) 50 held-out novel objects - Compute all metrics from the evaluation table in Section 2 - Compare against the baseline from Section 3.3

TODO: Implement Evaluation Pipeline

```
def evaluate_model(model: dict, eval_env, object_set: list,
                   num_episodes: int = 100) -> dict:
    """
    Evaluate the full model in simulation.

    Args:
        model: dict with 'encoder', 'rssm', 'flow_head'
        eval_env: PyBullet simulation environment with methods:
            - reset(object_name) -> (rgb, depth, instruction)
            - step(action) -> (rgb, depth, reward, done, info)
        object_set: list of object names to evaluate on
        num_episodes: number of episodes per object

    Returns:
        dict with:
            - "pick_success_rate": float
            - "mean_cycle_time": float (seconds)
            - "mean_smoothness": float (jerk metric)
            - "collision_rate": float
            - "per_object_success": dict mapping object name to success rate

    Implementation steps:
        1. For each object in object_set:
           a. Reset environment with that object
           b. Initialize RSSM state
           c. Loop until done or max 200 steps:
              - Encode observation
              - Update RSSM state (observe mode)
              - Build conditioning vector from RSSM state
              - Generate action chunk via flow_head.sample()
              - Execute first action from chunk in environment
              - Record metrics
           d. Log success/failure

        2. Aggregate metrics across all objects and episodes
        3. Print a summary table

    Hints:
        - Use action chunk overlap: generate a 16-step chunk but only
          execute the first 4 steps, then re-plan. This provides temporal
          consistency while allowing adaptation.
        - Compute jerk as the finite difference of acceleration:
          jerk = np.diff(actions, n=3, axis=0)
    """
    # YOUR CODE HERE
    raise NotImplementedError
```

3.7 Error Analysis

TODO: Failure Mode Analysis

```
def analyze_failures(eval_results: dict, model: dict, eval_env) -> dict:
    """
    Systematically categorize and analyze failure cases.

    From the evaluation results, identify all failed episodes and
    categorize them into failure modes.
    Expected failure categories:
        1. "grasp_slip" -- object was contacted but slipped during lift
        2. "miss" -- gripper did not contact the target object
        3. "wrong_object" -- grasped a different object than instructed
        4. "collision" -- collided with bin wall or other objects
        5. "timeout" -- exceeded maximum steps without completing task

    For each category:
        a. Count the number of failures
        b. Identify common object properties (size, shape, material)
           that correlate with this failure mode
        c. Visualize 3 example failures (save rgb frames at key moments)
        d. Suggest a specific mitigation strategy

    Returns:
        dict mapping failure category to:
            - "count": int
            - "fraction": float
            - "correlated_properties": list of strings
            - "mitigation": string describing fix

    Thought questions (answer in a markdown cell):
        1. Which failure mode is most common? Is this expected given the
           model architecture?
        2. Are there systematic differences in failure rates between
           suction and parallel-jaw grasps? Why?
        3. How does failure rate correlate with object size? With
           transparency? With deformability?
        4. If you could add ONE additional training signal to reduce
           failures, what would it be and why?
    """
    # YOUR CODE HERE
    raise NotImplementedError
```

3.8 Scalability and Deployment Considerations

TODO: Inference Benchmarking

```
def benchmark_inference(model: dict, device: torch.device,
                        num_warmup: int = 10, num_runs: int = 100) -> dict:
    """
    Profile inference latency for deployment feasibility.

    Measure the wall-clock time for each stage of the inference pipeline:
        1. Image preprocessing (resize, normalize)
        2. Observation encoding (DINOv2 + CLIP + cross-attention)
        3. RSSM state update
        4. Flow matching action generation (N=10 Euler steps)
        5. Total end-to-end latency

    Run each stage independently to isolate bottlenecks.

    Returns:
        dict with:
            - "preprocess_ms": mean preprocessing time in ms
            - "encode_ms": mean encoding time in ms
            - "rssm_ms": mean RSSM update time in ms
            - "flow_ms": mean flow matching time in ms
            - "total_ms": mean total time in ms
            - "throughput_hz": maximum achievable control frequency
            - "meets_latency_target": bool (total_ms < 200)

    Also compute:
        - Model parameter count per component
        - GPU memory usage during inference
        - Theoretical throughput on NVIDIA L4 vs T4 (Colab) vs A100

    Hints:
        - Use torch.cuda.synchronize() before timing to ensure accurate
          GPU measurements
        - Use torch.cuda.Event for precise GPU timing
        - Discard warmup runs (first 10) to avoid JIT compilation effects
        - Report mean, std, p50, p95, p99 latencies

    Thought questions:
        1. Which component is the bottleneck? How would you optimize it
           for production?
        2. What is the tradeoff between flow matching steps (quality) and
           latency? Plot this curve.
        3. Could you use model distillation or quantization to meet the
           latency target on cheaper hardware?
    """
    # YOUR CODE HERE
    raise NotImplementedError
```

3.9 Ethical and Regulatory Analysis

TODO: Ethical Impact Assessment

```
"""
Write your ethical impact assessment as a structured analysis.
For each category below, provide 3-5 sentences of analysis.

1. LABOR DISPLACEMENT
   - How many warehouse workers could this system displace at MegaMart?
   - What is the timeline for displacement?
   - What retraining or transition programs should NovaPick advocate for?
   - What is the net job impact (jobs displaced vs. new jobs created in
     robotics maintenance, supervision, ML engineering)?

2. SAFETY
   - What happens when the model makes a confident but wrong prediction?
   - How should the system handle uncertainty? (When should it ask for
     human help vs. proceed?)
   - What are the physical safety risks to humans working near these robots?
   - How does ISO 10218-1 compliance affect the system design?

3. BIAS AND FAIRNESS
   - The training data comes from specific robot platforms in specific labs.
     What biases might this introduce?
   - If the system performs worse on certain product categories (e.g.,
     products marketed to specific demographics), is this a fairness concern?
   - How would you audit the system for systematic performance disparities
     across product categories?

4. ENVIRONMENTAL IMPACT
   - Estimate the carbon footprint of training (128 A100s for 14 days)
   - Compare to the carbon savings from more efficient warehouse operations
   - Is the tradeoff justified?

5. DUAL USE AND MISUSE
   - Could this technology be misused for autonomous weapons or surveillance?
   - What safeguards should NovaPick implement?

Write your analysis below (minimum 500 words total):
"""
# YOUR ANALYSIS HERE (as a markdown cell)
```

Section 4: Production and System Design Extension

This section is designed for advanced students who want to understand how to take the model from a Colab notebook to a production deployment serving 14 fulfillment centers.

Architecture Diagram

```
                              NovaPick Production Architecture

+------------------+     +-------------------+     +------------------+
|   Robot Cell     |     |   Edge Compute    |     |   Cloud Backend  |
|                  |     |   (per cell)      |     |                  |
|  +------------+  |     |  +-------------+  |     |  +------------+  |
|  | RealSense  |--+---->|  | Preprocess  |  |     |  | Model      |  |
|  | Camera     |  |     |  | Pipeline    |  |     |  | Registry   |  |
|  +------------+  |     |  +------+------+  |     |  | (MLflow)   |  |
|                  |     |         |          |     |  +------------+  |
|  +------------+  |     |  +------v------+  |     |                  |
|  | Franka     |<-+-----+  | TensorRT   |  |     |  +------------+  |
|  | Controller |  |     |  | Inference   |  |     |  | Training   |  |
|  +------------+  |     |  | Engine      |  |     |  | Pipeline   |  |
|                  |     |  +------+------+  |     |  | (128xA100) |  |
|  +------------+  |     |         |          |     |  +------------+  |
|  | Safety     |  |     |  +------v------+  |     |                  |
|  | Controller |  |     |  | Action      |  |     |  +------------+  |
|  | (hardware) |  |     |  | Smoother    |  |     |  | Data Lake  |  |
|  +------------+  |     |  +------+------+  |     |  | (S3/GCS)   |  |
|                  |     |         |          |     |  +------------+  |
+------------------+     |  +------v------+  |     |                  |
                         |  | Telemetry   |--+---->|  +------------+  |
                         |  | Agent       |  |     |  | Monitoring |  |
                         |  +-------------+  |     |  | (Grafana)  |  |
                         +-------------------+     +------------------+
```

API Design

gRPC Service Definition (preferred over REST for latency):

```
service NovaPick {
  // Real-time action generation (streaming)
  rpc GenerateActions(stream ObservationRequest)
      returns (stream ActionChunkResponse);

  // Single grasp quality prediction
  rpc PredictGraspQuality(GraspQueryRequest)
      returns (GraspQualityResponse);

  // World model imagination for planning
  rpc ImagineTrajectory(ImaginationRequest)
      returns (ImaginationResponse);

  // Health check
  rpc HealthCheck(Empty) returns (HealthResponse);
}

message ObservationRequest {
  bytes rgb_image = 1;           // JPEG-compressed, 480x640
  bytes depth_map = 2;           // 16-bit PNG compressed
  string instruction = 3;        // Natural language
  repeated float proprioception = 4;  // 14 floats
  repeated float gripper_state = 5;   // 2 floats
  int64 timestamp_us = 6;        // Microsecond timestamp
}

message ActionChunkResponse {
  repeated float joint_velocities = 1;  // 16 x 7 = 112 floats
  repeated float gripper_commands = 2;  // 16 x 3 = 48 floats
  float grasp_confidence = 3;
  float latency_ms = 4;
  int64 timestamp_us = 5;
}
```

Serving Infrastructure

Component	Technology	Justification
Model serving	NVIDIA Triton Inference Server	Native TensorRT support, dynamic batching, model versioning
Model format	TensorRT FP16	2-3x speedup over PyTorch, fits L4 GPU memory
Edge compute	NVIDIA Jetson AGX Orin (64GB)	Per-cell deployment, low latency, sufficient for TensorRT inference
Orchestration	Kubernetes (K3s on edge)	Lightweight edge orchestration, rolling updates
Communication	gRPC with Protocol Buffers	~10x lower latency than REST/JSON for binary data
Image transport	Shared memory (between camera driver and inference)	Zero-copy image transfer eliminates serialization overhead

Scaling strategy: Each robot cell has its own Jetson AGX Orin running inference locally. The cloud backend handles training, model updates, and aggregate monitoring. Model updates are pushed to edge devices via OTA (over-the-air) updates during scheduled maintenance windows (2 AM - 4 AM).

Latency Budget

Stage	Budget	Actual (Target)
Camera capture + transfer	10ms	8ms
Image preprocessing (resize, normalize)	5ms	3ms
Observation encoding (DINOv2 + CLIP + cross-attention)	60ms	55ms
RSSM state update	15ms	12ms
Flow matching (10 Euler steps)	80ms	70ms
Action smoothing + safety check	10ms	5ms
Joint command transmission	5ms	3ms
Total	**< 200ms**	**156ms**
Safety overhead margin	44ms	--

Monitoring

Metrics to track (Grafana dashboards):

Metric	Alert Threshold	Dashboard
Pick success rate (rolling 1hr)	< 88%	Operations

Metric	Alert Threshold	Dashboard
Mean inference latency	> 180ms	Performance
P99 inference latency	> 250ms	Performance
GPU utilization	< 20% or > 95%	Infrastructure
GPU temperature	> 85C	Infrastructure
World model KL divergence (online)	> 2x training value	Model Health
Action chunk jerk (smoothness)	> 800 rad/s^3	Safety
Camera frame drop rate	> 2%	Sensor Health
Grasp confidence calibration error	> 0.15	Model Health
Items damaged per 1000 picks	> 5	Operations

Model Drift Detection

Strategy: Dual-signal drift detection

1. **Input drift:** Monitor the distribution of observation encoder embeddings using a reference distribution from the validation set. Compute the Maximum Mean Discrepancy (MMD) between the running embedding distribution (window: 10K observations) and the reference. Alert if MMD exceeds a threshold calibrated on the validation set.

2. **Prediction drift:** Track the calibration of grasp confidence predictions. Bin predictions into deciles and compare predicted success probability to actual success rate. If the Expected Calibration Error (ECE) exceeds 0.15, the model's confidence estimates are no longer reliable.

3. **Performance drift:** Track pick success rate with a CUSUM (cumulative sum) control chart. This detects sustained small shifts faster than simple threshold monitoring. A CUSUM alarm triggers model retraining.

Retraining trigger: Any two of the three signals exceeding thresholds within a 24-hour window triggers an automated retraining pipeline.

Model Versioning

Aspect	Approach
Registry	MLflow Model Registry with S3 artifact store
Naming	`novapick-v{major}.{minor}.{patch}` (semantic versioning)
Metadata	Training data hash, hyperparameters, eval metrics, training date
Rollback	One-command rollback to previous version; keep last 5 versions on edge devices
Promotion	`staging` -> `canary` -> `production` pipeline

Aspect	Approach
Immutability	Once registered, model artifacts are immutable; new version required for any change

A/B Testing

Framework: Each fulfillment center has 20-40 robot cells. For A/B tests:

- **Allocation:** Randomly assign 20% of cells to the challenger model (minimum 6 cells for statistical power)
- **Duration:** Minimum 48 hours, targeting 10,000 picks per variant
- **Primary metric:** Pick success rate
- **Guardrail metrics:** Damage rate (must not increase by > 0.2%), cycle time (must not increase by > 0.5s), collision rate (must not increase by > 0.5%)
- **Statistical test:** Two-proportion z-test with Bonferroni correction for multiple guardrails
- **Significance level:** alpha = 0.05, power = 0.80
- **Minimum detectable effect:** 1.5% absolute improvement in pick success rate
- **Auto-rollback:** If any guardrail metric degrades by > 2x the threshold during the test, automatically roll back the challenger

CI/CD for ML

```
Code Push → Unit Tests → Integration Tests → Training Pipeline
                                                      |
                                              Model Training
                                              (128x A100, ~3 days)
                                                      |
                                              Offline Evaluation
                                              (PyBullet sim, 10K episodes)
                                                      |
                                          ┌── Pass gates? ──┐
                                          |                 |
                                         Yes               No
                                          |                 |
                                    Register in          Notify team,
                                    MLflow               block promotion
                                          |
                                    Deploy to Staging
                                    (internal test cell)
                                          |
                                    24hr Staging Eval
                                          |
                                    Canary Deploy (5% cells)
                                          |
                                    48hr A/B Test
                                          |
                                    Full Production Rollout
```

Validation gates (must pass ALL to promote): 1. Pick success rate on novel objects > 88% in simulation 2. Pick success rate on known objects > 95% in simulation 3. Mean inference latency < 180ms on L4 GPU 4. No regression on any guardrail metric vs. current production model 5. World model imagination accuracy (cosine similarity) > 0.83

Cost Analysis

Training costs (per training run):

Resource	Quantity	Duration	Unit Cost	Total
A100-80GB (cloud)	128	14 days	$2.21/hr	$95,000
Storage (training data)	50 TB	Ongoing	$0.023/GB/mo	$1,150/mo
Storage (model artifacts)	500 GB	Ongoing	$0.023/GB/mo	$12/mo
Data pipeline (preprocessing)	32 CPU VMs	2 days	$0.50/hr	$768
Total per training run				**~$97,000**

Expected training frequency: quarterly (4x/year) = $388,000/year

Inference costs (per fulfillment center):

Resource	Quantity	Cost	Total/month
Jetson AGX Orin (amortized)	40 cells	$85/mo (3yr amortization of $2,999)	$3,400
Power (per Orin, 60W)	40 cells	$6.50/mo	$260
Edge networking	1 site	$500/mo	$500
Cloud monitoring + storage	1 site	$2,000/mo	$2,000
Total per center/month			**$6,160**

14 fulfillment centers: $86,240/month = **$1.03M/year** for inference

Total annual cost: ~$1.42M (training + inference for all 14 centers)

ROI comparison: MegaMart's current warehouse labor cost is $1.2B/year. A 30% reduction through automation saves $360M/year. NovaPick's system cost of $1.42M represents a **253x return on the compute investment** alone.

Section 5: Portfolio and Resume Packaging

5.1 Resume Bullet Points

1. Built a Vision-Language-Action model achieving 91% pick success rate on novel objects, improving zero-shot generalization by 56 percentage points over the classical grasp planning baseline, by implementing an RSSM world model with flow matching action generation trained on 15K real robot manipulation episodes.

2. Reduced end-to-end inference latency from 340ms to 156ms on edge hardware (NVIDIA L4 GPU) by optimizing the observation encoder with TensorRT FP16 quantization and implementing action chunk caching with 4-step execution windows.

3. Designed a production ML system for 14-site warehouse deployment serving 2.1M daily orders, incorporating gRPC streaming APIs, CUSUM-based drift detection, and automated A/B testing with guardrail metrics, reducing per-pick cost from $3.58 (manual) to$ 0.42 (automated).

4. Conducted systematic error analysis across 5 failure categories and 100 object types, identifying gripper mode selection as the primary failure driver and proposing a tactile-conditioned refinement module that improved success rate on deformable objects from 67% to 84%.

5.2 LinkedIn Project Description

Developed a world model and Vision-Language-Action system for autonomous robotic pick-and-place in high-SKU e-commerce fulfillment. The system combines a Recurrent State-Space Model for predicting grasp outcomes in latent space with a flow matching action generator that produces smooth robot trajectories conditioned on camera input and natural language instructions. Unlike classical grasp planning pipelines that require per-object engineering, this approach generalizes zero-shot to unseen objects by learning physical affordances from diverse manipulation data. Trained on 15K real robot episodes from the DROID and Open X-Embodiment datasets, the model achieves 91% pick success on novel objects -- a 56-point improvement over the analytical baseline. The production architecture uses edge inference on NVIDIA Jetson hardware with sub-200ms latency, supporting real-time 20 Hz control. System design includes drift detection, automated A/B testing, and a full CI/CD pipeline for continuous model improvement across a multi-site deployment.

5.3 GitHub README Template

```
# NovaPick: World Model + VLA for Autonomous Pick-and-Place

A world model and Vision-Language-Action system for zero-shot robotic
pick-and-place in high-SKU warehouse environments.

## Problem

E-commerce fulfillment centers handle 80,000+ SKUs that change quarterly.
Classical grasp planning requires per-object engineering (~$300/SKU) and
fails on novel items (34.7% success rate). We need a system that
generalizes to unseen objects without manual programming.

## Approach

We combine three concepts from recent world model research:

1. **RSSM World Model** (DreamerV3-inspired): Predicts grasp outcomes in
   latent space before physical execution, enabling mental rehearsal
2. **Multimodal Observation Encoder**: Fuses RGB-D vision with language
   instructions via cross-attention (DINOv2 + CLIP backbone)
3. **Flow Matching Action Generator** (pi-0-inspired): Generates smooth
   16-step action chunks at 20 Hz, replacing jerky single-step predictions

## Architecture
```

Camera (RGB-D) ──┐ ├──> Observation ──> RSSM World ──> Flow Matching ──> Robot Language Inst. ───┘ Encoder Model Action Head Actions (DINOv2+CLIP) (det+stoch) (10 Euler steps) (20 Hz)

~~~
## Results

| Metric | Baseline | Ours |
|---|---|---|
| Pick success (novel objects) | 34.7% | 91.2% |
| Pick success (known objects) | 96.2% | 97.8% |
| Cycle time | 3.1s | 3.4s |
| Inference latency | N/A | 156ms |
| Damage rate | 1.2% | 0.3% |

## Setup

```bash
# Clone repository
git clone https://github.com/username/novapick.git
cd novapick

# Install dependencies
pip install -r requirements.txt

# Download pretrained models
python scripts/download_models.py

# Download evaluation dataset
python scripts/download_data.py --split eval
~~~

Usage

```
from novapick import NovaPickModel, PyBulletEnv

# Load model
model = NovaPickModel.from_pretrained("novapick-v1.0.0")

# Create environment
env = PyBulletEnv(render=True)

# Run pick-and-place
obs = env.reset(object="red_mug")
while not done:
    action_chunk = model.predict(
        rgb=obs.rgb,
        depth=obs.depth,
        instruction="pick up the red mug"
    )
    obs, reward, done, info = env.step(action_chunk)
```

Project Structure

```
novapick/
  models/
    encoder.py         # Multimodal observation encoder
    rssm.py            # Recurrent State-Space Model
    flow_head.py       # Flow matching action generator
  data/
    dataset.py         # DROID + Open X-Embodiment data loading
    augmentation.py    # RGB-D + language augmentation
  eval/
    benchmark.py       # Simulation evaluation
    error_analysis.py  # Failure mode categorization
  deploy/
```

```
    triton_config/     # Triton Inference Server configuration
    edge_deploy.py     # Jetson deployment scripts
```

Future Work

- Tactile feedback integration for deformable object manipulation
- Multi-arm coordination for bin-to-box packing optimization
- Online adaptation via few-shot learning from deployment failures
- Sim-to-real transfer with domain randomization at scale ```

5.4 Interview Preparation

Question 1: "Why did you choose flow matching over a standard diffusion model for action generation?"

Why interviewers ask this: They want to see that you made deliberate architectural decisions rather than blindly following a paper. This tests understanding of generative model tradeoffs.

Key points for a strong answer: - Flow matching uses optimal transport (straight paths from noise to data), requiring fewer integration steps (10 vs. 50-100 for DDPM) -- critical for meeting the 200ms latency budget - Straight transport paths produce smoother action trajectories with lower jerk, which is important for robot safety and mechanical wear - Flow matching has a simpler training objective (MSE on velocity field) with no noise schedule to tune - Acknowledged tradeoff: diffusion models can produce more diverse samples, which matters less for action generation (we want the best action, not diverse actions)

Question 2: "How do you handle distribution shift when the system encounters objects it has never seen before?"

Why interviewers ask this: Distribution shift is the number one failure mode in production ML. This tests practical deployment awareness.

Key points for a strong answer: - The world model provides a built-in uncertainty signal: when the prior and posterior diverge significantly (high KL), the model is encountering something unfamiliar - Grasp confidence score provides a second signal -- if confidence drops below a calibrated threshold (0.6), the system can fall back to a conservative pre-programmed grasp or request human teleoperation - Production monitoring uses MMD on encoder embeddings to detect input drift across the fleet, triggering retraining before performance degrades - The system logs all low-confidence episodes for human review, creating a continuous improvement data flywheel

Question 3: "Walk me through your training data strategy. How do you deal with the fact that robot manipulation datasets are small compared to vision/language datasets?"

Why interviewers ask this: Data strategy is often more important than model architecture. This tests practical ML engineering judgment.

Key points for a strong answer: - Leveraged transfer learning: visual backbone (DINOv2) pretrained on 142M images, language encoder (CLIP) pretrained on 400M image-text pairs -- both frozen, so robot data only needs to train the fusion layers, RSSM, and flow head - Combined multiple public datasets (DROID: 76K episodes, Open X-Embodiment: 53K episodes) for diversity across robot morphologies and environments - Used aggressive data augmentation (color jitter, spatial transforms, depth noise) to multiply effective dataset size by ~10x - Synthetic data from PyBullet with domain randomization fills gaps in real data coverage, particularly for rare objects and edge-case configurations

Question 4: "Your system runs at 20 Hz. What happens if a single inference takes longer than 50ms (the control deadline)? How do you handle real-time guarantees?"

Why interviewers ask this: Real-time systems require different thinking than batch ML. This tests systems engineering depth.

Key points for a strong answer: - Action chunks provide temporal buffering: each inference generates 0.8 seconds of actions (16 steps), so a single slow inference does not cause an immediate control gap - The action smoother interpolates between chunks, so even if a new chunk is slightly late, the robot continues executing the blended trajectory - Hardware safety controller (separate from ML) runs at 1 KHz on the Franka's real-time controller and can halt the robot within 50ms independently of the ML pipeline - Monitoring tracks P99 latency, not just mean -- if P99 exceeds 250ms, the system alerts the operations team and can automatically reduce flow matching steps from 10 to 5 (trading quality for speed)

Question 5: "If you had 3 more months and unlimited compute, what would you do differently?"

Why interviewers ask this: This reveals depth of understanding and ability to prioritize. The best answers show awareness of the system's current limitations.

Key points for a strong answer: - Add tactile sensing: the biggest failure mode is grasp slippage on deformable/smooth objects. A tactile-conditioned refinement module that adjusts grip force in real-time would address the top failure category - Train a larger world model with video prediction capability (Genie-style) that can actually visualize predicted outcomes, enabling human-interpretable planning and better debugging - Implement online fine-tuning: the system currently retrains quarterly, but each fulfillment center has unique object distributions. Continuous online adaptation from deployment experience (with safety constraints) would improve per-site performance - Scale to multi-arm coordination: real warehouse packing requires coordinating multiple arms working in overlapping spaces, which introduces collision avoidance and task allocation challenges beyond single-arm pick-and-place

---

This case study was generated for educational purposes as part of the Vizuara AI curriculum. The company "NovaPick Robotics" is fictional. All technical specifications are realistic and grounded in current research and industry practice.