ChatGPT

Execution Plan: Vision-Language-Action Robotic Learning Environment

Overview and Objectives

This plan outlines a **vision-language-action (VLA)** robotic learning environment built in simulation, aimed at training an AI agent to perform manipulation tasks (e.g. pick-and-place, turning valves, rotating handles) via modern VLA techniques. The system will integrate: (1) a **simulation environment** (Three.js for 3D rendering with a physics engine) supporting realistic object interactions, and (2) a **VLA model** (based on **OpenVLA**) that perceives visual input and text instructions and outputs robot actions. The user will **demonstrate tasks using an Apple Vision Pro** (recording 3D hand motions and video), providing imitation learning data. The agent will be trained via imitation (with possible fine-tuning or reinforcement learning) primarily on the user's MacBook Pro (M4, 48GB RAM), using local resources as much as possible and leveraging cloud GPU only if necessary. The execution plan is organized into modular sections covering environment setup, data collection, model architecture, training procedure, and comparisons to alternative state-of-the-art methods.

Simulation Environment Setup (Three.js + Physics)

1. 3D Rendering (Three.js): Use **Three.js** to construct a virtual scene representing the robotic workspace. Three.js will handle real-time rendering of objects (tables, tools, valves, etc.) and the robot or end-effector. A virtual **camera** should be configured to simulate the robot's eye view or an external view of the scene (for training visuals). Ensure the camera's perspective and resolution match what the VLA model expects (e.g. images of a fixed size). The scene will include manipulable objects (blocks, levers, valves) with appropriate Three.js mesh geometries and materials for visual realism. Lighting can be added for clarity, but keep rendering simple enough for real-time performance on the MacBook.

2. Physics Engine Integration: Attach a physics engine to handle collisions, gravity, and joint constraints so that manipulation is realistic. Viable options (all JavaScript/WASM-based) include: **Cannon.js**, **Ammo.js**, or **Rapier.js**. Each provides rigid body dynamics and collision detection: - *Cannon.js:* A straightforward JavaScript physics library, but it is older and not well-maintained (only minor fixes via the community fork **cannon-es**). Cannon is known to be slower and may struggle with complex scenes [1], so it may not be ideal for high-fidelity simulation. - *Ammo.js:* A WebAssembly port of the Bullet physics engine. It supports advanced features (hinge joints for valves, sliders, etc.) and is fairly robust. Ammo.js can handle complex interactions (soft bodies, vehicles) if needed [2]. The trade-off is that it's heavier to run; however, given the MacBook's capability, Ammo is feasible for moderate simulations. - *Rapier.js:* A modern physics engine (Rust-based, compiled to WASM) that is **actively maintained and optimized**. Rapier offers excellent performance — benchmarks show it running **5–8× faster** than older libraries [3]. It supports rigid bodies, fixed joints, and articulations needed for our tasks. Rapier is a strong choice for speed and stability [4], especially if many objects or high physics frame rates are required.

Recommendation: Start with **Rapier.js** for physics due to its speed and active support. Use Cannon.js only for quick prototypes, and consider Ammo.js if specific Bullet features (e.g. complex joint types) become necessary.

3. Defining Task Environments: Set up separate scenes or configurations for each target task: - *Pick-and-Place:* Include a table or floor surface, objects to pick up (e.g. a cube or bottle), and a target location (another surface or container). The physics engine should give objects realistic mass and friction so they can be grasped and lifted. - *Valve Turning:* Model a valve as a rotatable wheel attached to a surface via a **hinge joint** (a physics constraint that allows rotation about one axis). For example, a cylinder or torus can represent the valve handle, and Rapier/Ammo can constrain it to one rotational degree of freedom (with some friction or resistance). - *Handle Rotation (Door handle or lever):* Similar to the valve, use a rigid body for the handle, attached with a hinge constraint. Configure limits if needed (e.g. a door handle might only turn 90 degrees). Ensure the handle's base (door or wall) is static.

The **agent's embodiment** in simulation can be represented in a simplified manner: - If using a **robot arm or gripper**, include a kinematic model of a gripper that can move and pinch. For simplicity, one might represent the end-effector as a sphere or small box (the "hand") that can be controlled in 3D space. The gripper "open/close" can be abstracted by an indicator or a distance between two small shapes. Collision layers can be set such that when the gripper closes around an object, the object can be "attached" (for example, temporarily disable gravity on a picked object or constrain it to the gripper). - Alternatively, since demonstrations come from a **human hand**, one could incorporate a **3D hand model** (e.g. a simple articulated hand) in the scene for more visual realism. However, simulating full human hand physics is complex; a simplified approach is to treat the demonstrator's hand as a kinematic controller that can apply forces to objects.

4. Physics Tuning: Configure the physics timestep for a stable simulation. A smaller timestep (e.g. 60–120 Hz physics update) yields more accuracy (important for grasp stability or smooth turning). Since the MacBook M4 is powerful, you can target 60 Hz rendering with physics steps at e.g. 120 Hz if needed for stable contact. If using Rapier, use its **fixed timestep** mode to avoid divergence. For Cannon/Ammo, tune solver iterations and collision tolerances if objects jitter.

5. Object Interaction Mechanics: Implement logic for *picking up objects*. In a simulated setting, a simple approach is: - When the agent's gripper (or hand proxy) is in contact with an object and a "close gripper" action is issued, create a temporary fixed constraint between the gripper and the object (making the object follow the gripper). This mimics a grasp. On "open gripper", remove the constraint so the object is released [5]. - Ensure the object's weight and the gripper's strength are such that lifting doesn't break the simulation (the object shouldn't be too heavy or the gripper too weak in the physics engine). Adjust friction so objects don't slip out too easily. - For turning a valve or handle, the agent's action will apply torque. For example, if the agent "grasps" the valve, you can read the agent's intended rotation from the action and directly apply a small incremental rotation to the valve via the physics API (or apply an impulse). The physics engine's joint will enforce realistic movement (e.g. requiring continuous contact to keep turning).

6. Camera Views and Recording: Add a virtual camera in Three.js to generate the image observations. For imitation learning, it's typical to use **egocentric view** (what the robot would see). In simulation, you could mount a camera at the eye position of a virtual robot or an overhead camera if that's the chosen perspective. The camera feed will produce an image each timestep; use Three.js render to canvas, then extract the pixel data. This can be done off-screen in Node.js or in a headless browser environment if

automating data collection. During training, each state's image (or sequence of images) will be input to the VLA model, so ensure consistent resolution (e.g. 224×224 pixels, which is used in OpenVLA's vision backbone [6]).

7. Simulation Control Interface: Provide a simple API to **step the simulation** given an action. For example:

```
function stepSimulation(action) {
    // action might be a discrete token or structured [Δx, Δy, Δz, Δroll,
Δpitch, Δyaw, grip]
    applyActionToEndEffector(action);
    physicsWorld.step(timeStep);
    renderScene();
    return captureCameraImage();
}
```

This function will be used both to play back demonstration actions in the sim (for validation) and later to let the trained agent interact with the environment. The action application logic should convert the model's output (tokens or values) into movements: e.g. "token 5 for X-axis" meaning move +5 cm in x direction, etc., as defined by your discretization (explained later). For rotation or valve turning, actions might directly correspond to small angle increments. Keep these increments small enough for smooth control (you can discretize the continuous action space into fine-grained bins, which the model will learn as tokens).

8. Task Reset and Episodes: Implement a way to reset the scene to initial conditions for each training episode. For example, after an attempt at pick-and-place, reposition the object at a start location (randomize it slightly for diversity), and reset the gripper to a home pose. Automate this so that you can generate many simulated episodes if needed (for reinforcement learning later or validation). In simulation, you can also randomize visual aspects (lighting, object colors, backgrounds) to improve generalization, though initially it can remain constant if demonstrations are all in one environment.

By following this setup, you will have a **web-based simulation environment** that can be controlled programmatically and provides the necessary data (image frames, physical state, etc.) for the learning pipeline. The next steps involve collecting demonstration data using Vision Pro and preparing the learning model.

Vision Pro Hand-Tracking Demonstrations

To teach the agent via **imitation learning**, you will collect expert demonstration trajectories using the Apple **Vision Pro** headset. The Vision Pro offers **inside-out hand tracking** and the ability to record the wearer's viewpoint, which we leverage to get paired video and 3D hand pose data for each task demonstration. Below is the plan for setting up demonstrations:

1. AR Setup for Demonstration: Develop a simple **visionOS app** or AR recording utility that places virtual task elements in the real world for the Vision Pro user to interact with: - Use **ARKit (visionOS)** to anchor virtual objects in the physical environment corresponding to the simulation setup. For instance, place a

virtual cube on a table in AR (for pick-and-place) or a virtual valve on a wall. This allows the demonstrator to see and manipulate virtual objects as if they were real. - Ensure the virtual objects in AR have the same spatial arrangement as in the simulation. This alignment is critical so that the recorded hand trajectories correspond to meaningful motions relative to the simulated objects. For calibration, you might define a coordinate frame (0,0,0) at a convenient reference (like the table center) and ensure both the AR scene and the Three.js simulation use that frame for consistency.

2. Hand Tracking Data Capture: Utilize **Vision Pro's hand tracking API** to record 3D hand joint positions over time: - ARKit provides anchors or a **HandTrackingComponent** that gives the positions of the hand and fingers [7]. You can track key points like the wrist, palm center, and fingertips. For grasping actions, you might specifically monitor the **index finger and thumb pinch** distance or a boolean "pinch detected" state (Vision Pro can detect pinch gestures). - Record the hand pose at each frame. For simplicity, you might record the 3D coordinates of the palm (or a proxy point between thumb and index if using pinch) and a binary flag for "gripper closed/open" (based on pinch or fist closure). This data will form the basis of the **action representation** for imitation: e.g. the change in hand position and the open/closed state.

3. Video Frame Recording: Simultaneously record the first-person video from the headset during the demonstration. The Vision Pro supports capturing **spatial videos** (3D videos) or screen recordings of the AR view [8]. In practice, you can start an AR recording via visionOS (possibly a built-in feature or programmatically by initiating a capture session). This will yield a video where virtual objects and the real background (and possibly the user's hand) are visible [8]. If possible, configure the recording to **60 FPS** for smoother alignment with tracked hand data (the hand tracking typically updates ~60 Hz as well).

4. Synchronization: It's crucial to sync the hand-tracking data with video frames. One approach is to use a common timestamp: - When recording, log the time for each hand tracking sample (ARKit will give timestamps with each update). - Later, you can sample the video frames by timestamp to align with the nearest hand data. If the AR recording includes the virtual objects, those objects' states (e.g. a highlight when grabbed) might also help verify alignment.

5. Data Storage Format: For each demonstration episode, save: - A sequence of image frames (extracted from the video). You might save these as a numbered image sequence or keep the video and note frame timestamps. - A corresponding sequence of hand poses/actions. For example, a JSON or CSV log of `t, hand_position (x,y,z), hand_rotation (if needed), pinch_state`. The **action** at time *t* could be defined as the difference in hand position from the previous timestep (Δx, Δy, Δz) plus the grip state. This naturally corresponds to the agent's action space (the 7-DoF delta mentioned in OpenVLA, which includes 3D translation, 3D orientation change, and gripper open/close [9]). If the demonstrator's wrist orientation is important (for valve turning, wrist rotation matters), also capture the hand's orientation (quaternion or Euler angles).

6. Number of Demonstrations: Collect multiple demonstrations for each skill: - Aim for a diverse set of at least **10–20 demos per task** to start (more if possible). For example, pick-and-place: demonstrate from different start positions and target placements; valve turning: demonstrate both clockwise and counter-clockwise turns, perhaps at different speeds or angles. - Variation is important: vary your approach slightly in each demo so the model doesn't overfit to one trajectory. The Vision Pro allows you to naturally vary motion (different angles of approach, etc., which is good).

7. Quality of Demonstrations: While recording, ensure the demonstrations are **successful and smooth**: - Move at a moderate pace (neither extremely slow nor too fast to blur the video). The model will learn the mapping of vision to actions; clear, deliberate movements help it infer the association. - For tasks that involve contact (picking or turning), make sure your virtual hand properly "grasps" the virtual object (from the AR perspective). You might implement a visual cue when a grasp is detected (e.g. highlight the object when pinch is held and the hand is overlapping it) to confirm that the demonstration logic considers the object grasped. - If mistakes happen (dropping an object, missing the handle), discard or redo that take, because inconsistent demonstration data can confuse training.

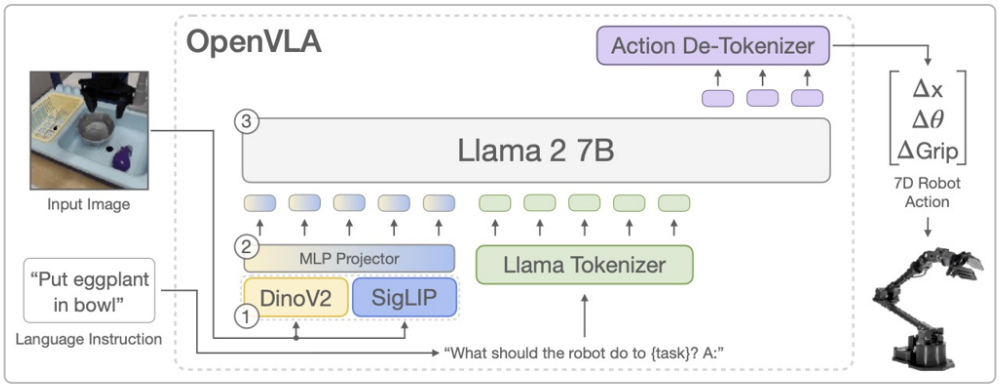
8. Data Post-processing: After recording: - **Extract frames** from the recorded video if needed (using a tool or script, e.g. `ffmpeg`, to get an image sequence). - Downsample or crop frames to the input size the model expects. If the Vision Pro FOV is wide, you may crop around the area of interest (or render virtual objects at a size where they roughly match how they'd appear in a fixed camera sim view). - **Coordinate normalization:** Convert the raw world coordinates of hand positions to the simulation coordinate frame. If you calibrated properly, this might just involve a consistent axis alignment and unit scaling. For example, 1 meter in AR might equal 1 meter in simulation units (Three.js often uses meters or arbitrary units). If not, apply a linear scaling factor and maybe a translation offset so that the demonstration positions align with how the simulation was set (e.g. ensure that when you reached for an object at AR (0.2,0,0.1) relative to table center, the simulation's object at (0.2,0,0.1) is indeed the same relative location). - **Action encoding:** Now convert the sequence of hand positions into action tokens or values. For imitation learning, typically we will train the model to predict the *next* action given the current image (and possibly instruction). So you might compute Δpos = hand_position(t+1) – hand_position(t) for each step, and use that (plus the gripper state) as the action at time t. These deltas can be continuous values, but since our model (OpenVLA) uses a discretized action space, you will **bin** these deltas into discrete tokens (e.g. small movements could be token "move +X small", etc., as described in the next section).

9. Example Data Structure: Each demonstration can be structured in an **RLDS (Robotic Learning Dataset)** format or simply as a sequence for sequence modeling: - **Language instruction:** A text describing the task, e.g. `"Pick up the cube and put it in the bowl."` or `"Turn the valve clockwise."` This will serve as the high-level command for the episode. - **Image frames:** e.g. `frame_000.png, frame_001.png, ... frame_N.png` – these are observations. - **Actions:** e.g. `action_000, action_001, ... action_{N-1}` – each action corresponds to transforming the environment from one frame to the next, as encoded by the demonstrator's hand movement. - We will use these (image, instruction) -> action pairs to train the model. OpenVLA's codebase supports mixing data from various sources if formatted properly [10]. You may create a custom script to feed this data into the training loop (for instance, by mimicking the format of Open-X Embodiment dataset entries [11]).

10. Validation in Sim: Before training the agent, it's useful to replay the demonstrations in the Three.js simulation to verify they make sense: - Take the recorded actions sequence and apply them in the simulation (using the `stepSimulation(action)` interface). Render the simulation frames and check that the virtual hand picks up the object, etc., according to the plan. This checks your coordinate alignment and action encoding. If something looks off (e.g. the simulated hand misses the object), adjust the coordinate transforms or action scaling.

By the end of this stage, you will have a **dataset of demonstration episodes**: each with an initial state and instruction, followed by a sequence of image observations and the demonstrator's actions. This dataset is the foundation for training the VLA model.

VLA Model Architecture (OpenVLA-based Design)



Base Vision-Language-Action model architecture (OpenVLA). The system uses (1) a fused visual encoder (e.g. DINOv2 ViT and SigLIP) to extract image features, (2) a projector that maps visual features into the input space of a language model, and (3) a 7B-parameter LLM (Llama 2 backbone) that outputs a sequence of tokens representing the action. The output tokens are finally decoded into continuous robot actions (e.g. 7-DoF end-effector motions and gripper command) [12] .

1. OpenVLA Overview: We adopt **OpenVLA** as the core model architecture. OpenVLA is a large multimodal policy that takes a **visual input** (camera image) and a **language instruction**, and produces a **robot action** as output [11] . Internally, it combines a **vision model** with a **language model** and an **action decoder**: - The **Vision Encoder** is a pretrained network (OpenVLA uses **DINOv2** and **SigLIP** backbones fused) that turns an input image into a set of visual feature embeddings [12] . In our setup, we can use the same encoders (ensuring our image size and normalization match their requirements) or substitute a lighter CNN if needed for local training (though using the pretrained ones is ideal for transfer learning). - The **Language Model (LLM)** is the core that processes the concatenation of language prompts and encoded visual features. OpenVLA specifically uses a Llama-2 7B model, which has been augmented to handle visual tokens [13] . This LLM is essentially the "brain" reasoning about "given what I see and the instruction, what should I do next?" - The **Action Decoder (Action Tokenizer)**: Instead of outputting free-form text, the LLM outputs a sequence of **action tokens**. These tokens correspond to a discretized representation of the robot's motor commands. In OpenVLA's design, the 7-DoF action (3D translation, 3D rotation, gripper open/close) is **normalized and discretized into tokens** [9] . For example, each degree of freedom might be binned into 256 possible values, and each value is represented by a token from a special action vocabulary. The sequence of tokens can then be decoded back into the numeric action values (e.g. token "X+ small" -> +0.01 m in x-axis, etc.). Finally, those numeric values are sent to the robot or simulator.

2. Customizing Action Space: We will define our own action tokenization scheme matching our simulation: - Each continuous parameter (like Δx movement or Δyaw rotation) will be quantized. Following OpenVLA's approach, a straightforward method is to allocate a fixed number of bins (e.g. 256) evenly across the range of each parameter [14] . For instance, allowable Δx might range from -5 cm to +5 cm; mapping that to 256 bins gives a resolution of ~0.04 cm per bin. Similarly for rotation (maybe -10° to +10° per timestep, discretized) and gripper (a binary or small set of tokens for open vs closed). - Each combination of these discretized values can be represented by a token (OpenVLA might use one token per DoF per step, or a compound token – but typically they flatten into a sequence). - **Learned Embeddings:** The model will have learned embeddings for each action token so that the LLM can treat actions analogously to words. Initially, since we are fine-tuning an existing model, we'll use its existing action vocabulary (OpenVLA's default) which

already has, say, 256 tokens per dimension. If our action ranges differ, we might reuse their scheme and scale our values to their normalized range (OpenVLA outputs normalized actions that are later un-normalized using training data stats [9]).

3. Vision and Language Input Formatting: For each step, the input to the model will include: - The **instruction prompt** in text (e.g. `"In: What action should the robot take to pick up the cube and place it in the bowl? \nOut:"` as used in OpenVLA examples [15]). The prompt format usually has an **"In:"** section describing the scenario or instruction and an **"Out:"** where the model is expected to output an action. - The **image features** from the current frame. In practice, OpenVLA's `AutoProcessor` handles combining the image and text: it might insert special tokens representing the image patches for the LLM to attend to [16]. - (Optionally, one could also provide the previous action or state if doing sequence modeling; however, many VLA policies predict action from the current image+instruction only, implicitly assuming Markovian state. OpenVLA, being a transformer, can potentially consider previous outputs if fed in sequence, but a simpler approach is one-step prediction iteratively.)

4. Model Training Approach: We will **fine-tune** the OpenVLA model on our custom dataset of demonstrations (rather than train from scratch) because: - OpenVLA is already pretrained on a large corpus of robot data (970k real robot episodes) and has strong generalist capabilities [13] [17]. Fine-tuning will adapt it to our specific tasks (pick-place, valve-turn) and the simulation domain, leveraging its prior knowledge (e.g. understanding of language and basic physics). - Fine-tuning is feasible on a consumer machine using parameter-efficient methods. The OpenVLA authors showed that methods like **LoRA (Low-Rank Adaptation)** can be used to fine-tune the 7B model on **consumer GPUs** (like a single 24 GB GPU) and even quantize the model for efficiency without significant performance loss [18]. On a MacBook Pro with 48GB unified memory, using 4-bit quantization and LoRA to train only small adapters is a likely approach to get training running locally.

Training will use an **imitation learning objective**: essentially **next action prediction**. Formally, we minimize cross-entropy loss between the model's predicted token sequence and the ground-truth action tokens from the demonstration, at each time step. Since this is like sequence-to-sequence learning, we can use teacher forcing during training (feeding the actual previous action or just resetting at each step with the current image, depending on implementation). The OpenVLA codebase or Hugging Face's Transformers `VisionEncoderDecoderModel` (or `AutoModelForVision2Seq`) can be employed for this: - We will load the pretrained weights (`openvla/openvla-7b` on Hugging Face) [19]. - Freeze most of the model and attach LoRA adapters to the LLM layers to fine-tune with low memory. Alternatively, OpenVLA's provided training script can handle this (they have examples for LoRA fine-tuning [20]). - Input data: for each training sample, we provide the prompt + image, and expect the model to output the action token. If we feed entire sequences, we might concatenate a sequence of (image, action) pairs as one long sequence per episode (with special tokens separating steps). However, a simpler method is single-step training: treat each (image, instruction) with the next action as one training sample (the instruction remains the same throughout an episode, so it can be included each time or encoded once and cached). OpenVLA's fine-tuning on BridgeData tasks suggests it predicts one action at a time and was run in an autoregressive loop for multi-step execution [21].

5. Running on Mac (Metal Performance Shaders): Apple's M-series GPU can be used via PyTorch's MPS backend. Verify that the versions of PyTorch and Transformers support MPS for all operations (as of PyTorch 2.x, transformer operations on MPS are largely supported). If some operations are not optimal on MPS, an

alternative is to use the 8-core CPU with 48GB RAM, though that will be slow – so prefer the GPU: - Set `torch.device("mps")` and move model and data to it. Monitor memory usage; 7B parameters in 16-bit require ~14GB, which is within 48GB but leave headroom for activations. Using bfloat16 or mixed precision can help. The `AutoModelForVision2Seq.from_pretrained(..., low_cpu_mem_usage=True, torch_dtype=torch.bfloat16)` as shown in OpenVLA docs is useful [22]. - If memory is still an issue, quantize the model (there are 4-bit quantization methods or use a smaller model variant if available, e.g. an OpenVLA 1B model, or use **SmolVLA** which we'll discuss later).

If local training proves too slow or memory-intensive: - Use the **fallback to cloud**: e.g. spin up a GPU instance (with an NVIDIA A10 or A100 which has ample memory) on a service like AWS, or use Google Colab Pro which gives access to 24GB GPUs. The dataset is not huge (tens of demonstrations), so uploading it is fine. The OpenVLA code and model weights (7B) will need to be available; Hugging Face transformers can download the model if internet is available or you can copy the checkpoint to the instance. Ensure to still use parameter-efficient fine-tuning to keep costs low (one can fine-tune 7B in a few hours on a single A100 for such imitation learning tasks).

6. Training Procedure and Schedule:
- **Epochs and iterations:** Because imitation datasets are usually small, you might train for many epochs over the same data. Monitor the loss to avoid overfitting. For example, with ~50 demonstration trajectories, you might run 100 epochs and see the loss plateau. If overfitting (loss goes to near 0 and policy just mimics without generalizing), consider mild regularization or data augmentation (randomize backgrounds in images, etc., to improve robustness). - **Validation:** Ideally, set aside one or two demos as a validation set to ensure the model is learning generally (the success on a held-out demo or a variant task). You can also validate by executing the policy in the simulation: after certain training epochs, take the model and run it in the sim (feeding the initial image and instruction, then autoregressively feeding its predicted action back in to get next image, etc.). Check if it accomplishes the task. This "closing the loop" is important because a model can predict well one-step actions but might accumulate error over a multi-step sequence if it hasn't seen its own mistakes. If issues arise (e.g. model veers off), consider training with longer sequences or using approaches like **DAPG** (Dataset Aggregation) where you augment training data with some corrections.

7. Execution of Trained Policy: Once the model is trained, integrate it with the simulation for deployment: - Load the fine-tuned model on the MacBook (quantized for faster inference if possible). - For a given new instruction (say, "Rotate the valve handle"), do as OpenVLA does: capture the initial image from the sim, form the prompt, run `model.predict_action(...)` which will output the first action token [15]. Execute that action in the simulation, obtain the next image, and feed it in (with the same original instruction, possibly concatenated with some history if needed) to get the next action. Loop until the task is done (or a max number of steps). - Because this is an closed-loop execution, ensure the model inference is reasonably fast. On CPU 7B model would be slow (~seconds per step); on the M4 GPU or a smaller model it should be faster. OpenVLA with FlashAttention, etc., can achieve decent inference speed [23], but if that's insufficient, consider the alternatives below (like smaller SmolVLA or the FAST tokenization to speed up).

So far, the plan uses **OpenVLA as the baseline architecture** due to its strong performance and open-source availability. In summary, OpenVLA gives us a **LLM-driven policy** that can be fine-tuned to map vision & language to actions, with a discretized action space suitable for our tasks. Next, we provide comparative insights on alternative modern architectures (diffusion models, smaller models, high-frequency controllers) that could be considered and how they differ, to inform potential extensions or substitutions in the design.

Comparative Model Options and Alternatives

The field of robot learning is rapidly evolving. Below, we outline **modern VLA architectures** beyond our chosen baseline, including their approach, strengths, and applicability to this project. This will help in deciding if any alternative should be incorporated or if future upgrades are warranted. A summary comparison is also provided in tabular form for clarity.

Diffusion-Based Control Policies (π0, DreamVLA)

π0 (Pi-Zero): π0 is a **foundation model for robotics** introduced by the Physical Intelligence team (2024) and is one of the first generalist VLA models using *diffusion* techniques [24] [25]. Instead of outputting one token at a time autoregressively, π0 generates entire action trajectories by leveraging a **flow-matching diffusion model** [26]. The model gradually refines a random noise into a sequence of low-level motor commands, effectively "denoising" to a smooth trajectory that accomplishes the task. This enables **real-time 50 Hz control** output – the model can output actions at 20ms intervals, which is much faster than standard transformer policies [26]. The benefit is **very smooth and precise motions** (critical for dexterous tasks like folding cloth or precise grasps). π0's architecture still uses a vision-language backbone to understand high-level context, but the action generation is continuous and iterative (diffusion).

In our setup, π0's approach could yield smoother control for things like turning a valve (ensuring continuous rotation rather than stepwise motion). However, diffusion models are **computationally heavier** to train and run (multiple neural steps per action). Physical Intelligence addressed this with flow-matching (a training technique to speed up diffusion policy generation) and also introduced **π0-FAST**, an autoregressive version using the FAST tokenizer [27]. **π0-FAST** uses a similar token compression idea as OpenVLA's updates, achieving ~5× faster training than diffusion while maintaining performance [28]. If real-time fine control is a priority, one could explore π0 or π0-FAST open-source implementations (they have a JAX code and a Hugging Face port in the *LeRobot* repo [29] [30]). That said, running π0 fully on a MacBook might be challenging due to model size (it's on the order of billions of params as well) and the diffusion process. It might be more realistic to fine-tune π0 on cloud GPUs or stick with π0-FAST for local use, which aligns with the token-based approach we already have.

DreamVLA: DreamVLA (2025) is another cutting-edge approach where diffusion models are used in the context of VLA, specifically integrating **world-model "dreaming" with action**. DreamVLA introduced a **diffusion-based large language model (dLLM)** that can predict future events or "imagine" outcomes, which helps it plan inverse dynamics for control [31]. In essence, DreamVLA first uses a generative visual model (Dream-VL) to forecast what future states should look like, then uses that in a loop to decide actions – a kind of model-predictive control with a learned model. This approach yielded a high success rate (about **76.7% success on real robot tasks** in experiments, outperforming prior methods) [32]. For our purposes, DreamVLA's techniques might be beyond scope to implement from scratch, but we can take inspiration: the idea of imagining future frames could improve performance in long-horizon tasks (e.g. turning a valve to a specific angle might benefit from predicting when the valve will be sufficiently rotated). If interested, one could incorporate a simpler world-model (like train a small video predictor of the valve turning) and feed that into the policy. However, given the complexity, DreamVLA is more of a research frontier – we will likely stick to direct policies. It's worth noting DreamVLA and similar works are open-source (the paper and even a Hugging Face model for Dream-VLA-7B exist [33]), but require significant compute to utilize fully.

Pros & Cons for Our Setup: Diffusion-based models (π0, DreamVLA) excel in generating fine-grained, continuous actions and handling **multi-modal distributions** (they can naturally represent uncertainty and multiple modes of action). If our tasks were extremely dexterous or needed very **fluid motion** (like complex two-arm coordination or deformable object manipulation), these might be necessary. In our scope (single-arm, fairly structured tasks), the autoregressive OpenVLA approach is adequate and much simpler to deploy. Therefore, while we acknowledge π0 and DreamVLA, we opt for the simpler scheme but can consider adopting elements like FAST tokenization (already in OpenVLA's roadmap) or high-frequency control if needed later.

Lightweight & Efficient Models (SmolVLA and Others)

SmolVLA (450M): SmolVLA is a **small-scale VLA model (450 million parameters)** introduced by Hugging Face's LeRobot team in 2025. It is explicitly designed for **affordable and efficient robotics**, able to run on consumer hardware like a MacBook or even a 4GB VRAM GPU [34]. Despite its size, SmolVLA reportedly *outperforms larger models* on several robotics tasks by leveraging community datasets and efficient training tricks [35]. The key to SmolVLA's efficiency is in both architecture and training: - It likely uses a smaller language model backbone (perhaps a distilled LLM or a specialized policy network) combined with a compact vision encoder. - It also introduced an **asynchronous inference stack** [36]: separating the processing of vision and actions so that while one action is executing, the perception for the next step is already underway. This parallelism lets robots respond faster to changes because the visual processing doesn't stall the action execution loop [36].

For our project, SmolVLA offers an attractive fallback if OpenVLA-7B proves too heavy. We could use SmolVLA's architecture to train from scratch or fine-tune if a pretrained checkpoint is available. Its **450M size can easily fit in 48GB RAM** (even without quantization) and training it locally is more feasible (Hugging Face indicated it's trainable on a single GPU) [37]. SmolVLA is open-source and available on Hugging Face [38], making it easy to experiment with. The trade-off is potentially lower capacity: it might not handle as complex language understanding as a 7B LLM. However, for relatively simple instructions ("pick up the red cube and turn the valve"), 450M might suffice, given it's specialized for robotics and has strong performance per the TechCrunch report [35].

We can consider a two-phase approach: start developing with SmolVLA for rapid iteration (fast training/inference on Mac), then switch to OpenVLA or another larger model for maximum generalization once the pipeline works. Notably, SmolVLA's performance suggests it can generalize well with modest data; one user fine-tuned it with just 31 demos to control a robot arm, and it matched/outperformed single-task policies [39] – this implies it's data-efficient, which is great when we have limited demonstrations.

Other lightweight options include **RT-1** style models (Google Robotics Transformer) but those aren't openly available, or **smaller LLMs** (like a 3B parameter model) combined with CLIP vision. Given SmolVLA is purpose-built for this scenario, it's the top lightweight alternative.

High-Frequency Dual-Controller Architectures (Helix, GR00T N1)

For more advanced deployments, especially involving **complex robots or safety-critical, fast reactions**, recent architectures split the workload into two systems: - **System 2:** a high-level reasoning module (often a large VLM or LLM) operating at a low frequency (a few Hz). - **System 1:** a low-level controller (smaller, specialized network) operating at high frequency (tens to hundreds of Hz) to execute fine motions.

Figure's Helix: Helix (announced Feb 2025) is a VLA model designed for a **humanoid robot's full upper body control**, using the System1/System2 split [40] [41]. Helix's System 2 is an open 7B VLM (similar scale to OpenVLA's brain) that does scene and language understanding at ~7–9 Hz [42]. System 1 is an **80M-parameter transformer policy** (small enough to run on-device at 200 Hz) that takes the semantic output of S2 and produces continuous joint commands in real time [42] [43]. They train both systems end-to-end, so S1 learns to interpret S2's latent goals and handle the precise motions (like adjusting arm trajectory on the fly). A key point is Helix outputs **continuous actions directly** – it avoids discrete tokenization, which they found doesn't scale well to the 20+ DoF of a humanoid hand [44]. This makes sense: tokenizing a very high-dimensional action could explode the sequence length or lose nuance, whereas a learned controller can output joint torques or velocities directly. Helix demonstrated impressive generalization (picking up novel objects by language command) and multi-robot collaboration [45] [46], all running **fully on the robot (no external compute)** [47].

For our project, Helix represents a design philosophy rather than something we'd implement fully (Helix is specific to Figure's humanoid). However, the **principle of decoupling planning and control** could be applied in the future. If we find that OpenVLA (a single system) struggles to both decide *and* execute fine motions (for example, maybe it hesitates or isn't smooth due to the fixed time-step nature of token outputs), we could introduce a secondary control loop. For instance, the OpenVLA policy could output a target or waypoint, and a small **PID controller or learned network** could run at a higher rate to actually move the gripper to that waypoint, handling corrections in between model inference steps. This is a simplified version of System1/2: use the AI for high-level decisions, and a classical or small net for interpolation. In simulation at 60 Hz, our current model might suffice, but if tasks become dynamic (imagine catching a moving object or coordinating two arms), a Helix-like dual system is very relevant.

NVIDIA's Isaac GR00T N1: Announced in Mar 2025, GR00T N1 is an **open foundation model for humanoid robots** from NVIDIA, also using a dual architecture [48] [49]. It combines an **"Eagle" VLM + SmolLM (1.7B)** for the high-level part and a **diffusion transformer** for low-level motor commands [50] [51]. The System 2 (though NVIDIA's terminology swapped numbering) handles vision-language understanding and plans actions (this can even run offboard on the cloud to allow more compute) [52] [53]. System 1, the diffusion action model, runs on the robot to produce smooth motions with each step conditioned on the previous (hence a transformer-diffusion hybrid) [54] [51]. GR00T's highlight is **cross-embodiment generality** – one model works for many robot shapes without retraining [55]. They achieved this by massive training on both **synthetic data (750k simulated trajectories) and real human motion data** [56]. Notably, NVIDIA made GR00T N1 (2B parameters) openly available, with code and pretrained weights on Hugging Face [57]. This means if one wanted, they could download GR00T N1 and try it in our environment. However, 2B plus diffusion-based generation might still be heavy for local execution (though 2B is not too large; it might run on the Mac GPU at lower speeds).

For our needs, GR00T's relevance would be if we aim to support multiple robot embodiments or need the diffusion accuracy for motion. Given we focus on one simulated arm/hand, GR00T is probably overkill. But it's good to know that **pretrained high-frequency controllers exist and are open-source**. For example, one could experiment with GR00T's low-level diffusion controller, feeding it high-level goals from our system. This is complex, but as a thought: since GR00T was trained with lots of synthetic data (which presumably includes tasks like pick-place), its System1 might already know how to perform smooth pick and place if given the right intermediate goal. Integrating that would take significant work, so this is more of a reference point.

Comparison of VLA Model Options

For a clear summary, the table below compares these architectures on key aspects:

Model	Approach	Size (params)	Key Features / Notes
OpenVLA	Vision+LLM, discrete action tokens	**7B**	Open-source 7B VLA (Llama2-based); strong multi-task manipulation performance out-of-the-box [13] [58]. Uses DINOv2 + SigLIP vision; outputs 7-DoF normalized actions via 256-bin tokenization [12]. Fine-tunable on consumer hardware via LoRA [18].
π0 (Pi-Zero)	Diffusion (flow-matching) VLA	≈ *billions*	Generalist policy by Physical Intelligence (2024). Uses diffusion/flow-matching for continuous 50 Hz control [26], enabling very smooth trajectories. Trained on 7 robot types, 68 tasks [59]. Open-sourced (JAX code) but heavy to run; π0-FAST variant provides a faster, tokenized mode [27] [28].
DreamVLA	Diffusion-based LLM (dVLM) + control	**7B** (VLA)	2025 model integrating a diffusion language backbone for planning [31]. "Dreams" of future states to inform actions (world-model integration). Achieved ~76.7% success on real robot trials [32]. Open research model (available on arXiv/HuggingFace). High complexity, suited for research on foresight in policies.
SmolVLA	Vision+LLM, discrete actions	**450M**	Hugging Face's efficient VLA (2025) [34]. Tiny yet competitive model; **runs on MacBook** and low-cost GPUs [36]. Trained on community demos. Supports asynchronous vision-action pipeline for responsiveness [36]. Ideal for prototyping and low-resource deployment; could be fine-tuned quickly with our demos.
Helix	Dual-system: 7B VLM + 80M controller	**7B + 0.08B**	Figure's humanoid VLA (2025). **System 2:** 7B open VLM at 7–9 Hz for perception/language [42]. **System 1:** 80M transformer at 200 Hz for reactive control [42] [43]. End-to-end trained; outputs continuous joint commands (no tokenization) [44]. Excels at bimanual tasks, on-the-fly adaptation. Not publicly available weights (Figure uses it in-house); conceptually valuable for tasks needing real-time dexterity.

Model	Approach	Size (params)	Key Features / Notes
Isaac GR00T N1	Dual-system: VLM + Diffusion ctrl	**2B** (total)	NVIDIA's open generalist model (2025) [49] [51]. **High-level:** Eagle VLM + SmolLM (1.7B) for reasoning. **Low-level:** diffusion transformer for precise actions [51]. Trained on massive synthetic + real data, adaptable to various robots [55] [56]. **Publicly released** on HuggingFace [57]. Could be leveraged for its robust controller; however, diffusion controller may be slower to run without powerful GPUs.

(References in the table: OpenVLA details from [12] [58], π0 from [26], π0-FAST from [28], DreamVLA from [60], SmolVLA from [34] [36], Helix from [42] [44], GR00T N1 from [51] [56].)

Applicability to Our Project

- **Current Choice:** For the immediate implementation, **OpenVLA (7B)** fine-tuned with our data is a solid choice, balancing capability and existing support. If we encounter computational limits, we'll pivot to **SmolVLA (450M)** for faster iteration. Both follow the same paradigm of vision+LLM+token actions, which aligns well with our design and can be run on local hardware with adjustments.
- **Diffusion Models:** We may not incorporate π0 or DreamVLA initially due to complexity. However, their success suggests that if our agent struggles with the granularity of control (e.g. jerky motions from discrete actions), exploring diffusion-based fine control could help. One intermediate idea is using **Diffusion Policy** as a from-scratch imitation learner on our data (Diffusion Policy is an algorithm for training diffusion models on motion data). OpenVLA's paper actually compared to Diffusion Policy and found OpenVLA fine-tuned outperformed it on language-grounded tasks [61]. So for primarily language-conditioned tasks, the LLM approach might be superior, whereas diffusion shines in precise continuous control.
- **High-Frequency Controllers:** Our tasks (pick/place, turn valve) are relatively low-frequency motions; a single system operating at ~5–10 Hz might be enough (the Franka arm was controlled at 5–15 Hz in their experiments and succeeded [62]). Thus, we don't strictly need Helix/GR00T style split. But if later the project extends to say **teleoperating in real-time** or handling moving objects (where a 200 Hz reflex would be useful), we might add a secondary control loop. The asynchronous inference of SmolVLA already is a step in that direction, allowing overlapping compute and action which effectively increases responsiveness [36]. We could also script a simple high-frequency PID to refine positions as mentioned. For now, the plan keeps one policy loop for simplicity.

In conclusion, the chosen approach (OpenVLA with discrete actions) is validated by recent research as an effective solution for general manipulation [63] [64]. Alternatives like π0 and Helix represent the frontier which we can draw from if needed – π0's FAST tokenizer is already being folded into OpenVLA developments (noted as the **FAST action tokenizer** released by PI, giving 15× speedups in inference with compressed tokens [65]), and Helix's continuous control approach might inform how we design the gripper control (ensuring we don't overly quantize the gripper rotation, for example). We will keep an eye on these developments as the project progresses, but will proceed with the implementation using the plan outlined.

Implementation Steps and Timeline

Finally, we break down the execution into concrete steps with an indicative timeline:

Phase 1: Environment & Data (Weeks 1-2) - *Setup Three.js Simulation:* Develop the Three.js scene and integrate Rapier.js physics. Test basic object interactions (e.g. dropping an object, pushing it) to ensure stability. - *Vision Pro App:* Build a simple visionOS app for hand tracking and AR recording. Verify that we can log hand coordinates and capture video. (If development is time-consuming, alternatively use the Vision Pro's built-in capture and a separate script for hand logging, then sync via timestamps). - *Collect Demonstrations:* Record a batch of demonstrations for each task. Start with one task (e.g. pick-and-place) to refine the process, then do others. Aim for at least 10 good demos per task. - *Data Processing:* Write scripts to extract frames and align with actions. Visualize a couple of processed demos by overlaying the hand trajectory on frames or replaying in sim, to ensure data integrity.

Phase 2: Model Preparation (Weeks 3-4) - *Environment for Training:* Set up PyTorch with MPS on the Mac. Download OpenVLA 7B weights from Hugging Face [66]. Test loading the model and running a forward pass with dummy data (to ensure MPS/CPU can handle it). If issues, try quantization (use `bitsandbytes` 4-bit or FP16 with reduced precision). - *Fine-tuning Pipeline:* Use the OpenVLA codebase or Hugging Face `Trainer` to configure training. If using OpenVLA's repo, prepare a custom dataset loader for our demonstrations (possibly converting to their TFRecords or just writing a small Python dataset class that yields image tensor, prompt, and action token). Ensure that the action tokenization aligns with the model's expectations (we might use their existing tokenizer which handles images and text; we'll extend it to also handle our action labels). - *Run a Small Test:* Fine-tune on a very small subset (or even one batch) just to verify the loss is decreasing and the model can overfit a tiny example (as a sanity check). This can be done on CPU overnight if needed or quickly on a cloud GPU.

Phase 3: Training & Evaluation (Weeks 5-6) - *Full Fine-Tuning:* Launch the fine-tuning on the Mac (or cloud if needed). Monitor GPU memory and utilization. Use callbacks to save checkpoints periodically. - *Evaluation in Simulation:* After training, load the fine-tuned model and perform rollouts in the sim for each task: - For instance, reset the scene to initial state, feed the instruction, then loop: get image -> model predicts action -> apply action -> repeat. Measure success (did the cube end up in bowl? Did the valve turn fully?). Because these tasks have clear end conditions (object in place, valve at angle), we can script success checks. - If the agent fails or stalls, diagnose whether it's a modeling issue (e.g. not enough understanding) or a data issue. We might collect additional demos to cover failure cases and fine-tune further (this is like iterative learning). - *Optimize:* If inference is slow (say >1s per step), implement optimizations: enable FlashAttention (OpenVLA already supports it [23]), or offload model to a faster machine for execution. If the policy acts but in a jittery way, consider enabling the FAST tokenizer compression [67] to reduce sequence length, or minor smoothing on action outputs.

Phase 4: Extensions & Options (Week 7+)
- *Try SmolVLA:* As an experiment, fine-tune SmolVLA on the same data and compare performance. Since SmolVLA can train faster, this could be done in parallel. Evaluate if the smaller model can achieve similar success. This gives a sense of how important the large LLM is for these tasks. - *Integrate High-frequency Control:* If the actions from the model are not fine enough (e.g. overshooting a position), integrate a secondary controller. This could be as simple as: after each model action, do a few physics sub-steps where you smoothly interpolate the hand towards the target (dampening oscillations). Or use a little neural network that was trained to adjust errors (we could even take the demonstration data to train a network

that given the current vs desired position outputs a correction – essentially behavioral cloning for a local policy). - *Cloud Training if Adding Data:* If we gather significantly more data (say dozens of demonstrations or additional tasks), and want to perhaps fine-tune a larger model variant (like OpenVLA-13B if it existed), consider using a cloud GPU cluster. At that stage, one might integrate the pipeline with **Ray Tune or Azure** to do hyperparameter sweeps (finding the best learning rate, etc.). - *Real-World Transfer:* Though beyond the current scope, a natural extension is to deploy the trained policy on a real robot or in AR. We could test the policy by **feeding real camera images** (for instance, from the Vision Pro or an iPhone camera) to the model to see if it still predicts reasonable actions. If the simulation visuals are close to real (domain randomization helps here), the model might zero-shot generalize to the real world (OpenVLA has shown strong sim2real and real2real transfer abilities when fine-tuned [64] [68]). If not, one could collect a few real-world demos and fine-tune further.

Throughout development, maintain clear documentation and modular code: - The simulation environment (Three.js) and the machine learning model training (Python) are separate; define a clear interface for data exchange (likely just reading/writing demonstration files and perhaps a socket or REST API if you ever want the model to control the sim live). - Use version control (Git) for code and keep track of model checkpoint versions.

By following this plan, the user will build a fully functional VLA-based robotic learning environment where they can prototype new tasks quickly in simulation, demonstrate them via AR, and train an AI agent locally. This system leverages state-of-the-art research (OpenVLA and related models) while remaining practical for a MacBook setup, and is extensible to incorporate newer techniques like diffusion controllers or dual systems if the need arises.

References: This plan was informed by the OpenVLA open-source model and paper [13] [12], recent advances in diffusion policies like π0 [26], efficiency innovations in SmolVLA [34], and emerging dual-system architectures Helix and GR00T [42] [51], among other sources. These references ensure the blueprint aligns with current best practices in robot learning and can be trusted as a foundation for development.

---

[1] [4] Rapier vs Cannon performance? - Questions - three.js forum
https://discourse.threejs.org/t/rapier-vs-cannon-performance/53475

[2] Choosing a Physics Engine for Your 3D Browser Game
https://3dgame.app/blog/physics-engines-comparison

[3] Announcing the Rapier physics engine - Dimforge
https://dimforge.com/blog/2020/08/25/announcing-the-rapier-physics-engine/

[5] [6] [12] [17] [58] [61] [62] [64] [68] OpenVLA: An Open-Source Vision-Language-Action Model
https://openvla.github.io/

[7] Tracking and visualizing hand movement - Apple Developer
https://developer.apple.com/documentation/visionOS/tracking-and-visualizing-hand-movement

[8] Take a capture or recording of your view on Apple Vision Pro
https://support.apple.com/guide/apple-vision-pro/take-a-capture-or-recording-of-your-view-tan1527c9e00/visionos

[9] [11] [16] [21] [66] openvla/openvla-7b · Hugging Face
https://huggingface.co/openvla/openvla-7b

10 14 15 19 20 22 23 65 67 GitHub - openvla/openvla: OpenVLA: An open-source vision-language-action model for robotic manipulation.
https://github.com/openvla/openvla

13 18 63 [2406.09246] OpenVLA: An Open-Source Vision-Language-Action Model
https://arxiv.org/abs/2406.09246

24 Policies in AI Robotics - phospho starter pack documentation
https://docs.phospho.ai/learn/policies

25 Vision Language Action Models (VLA) & Policies for Robots
https://learnopencv.com/vision-language-action-models-lerobot-policy/

26 27 28 29 30 59 π0 and π0-FAST: Vision-Language-Action Models for General Robot Control
https://huggingface.co/blog/pi0

31 [2512.22615] Dream-VL & Dream-VLA: Open Vision-Language and ...
https://arxiv.org/abs/2512.22615

32 60 DreamVLA: A Vision-Language-Action Model Dreamed ... - NeurIPS
https://neurips.cc/virtual/2025/poster/118226

33 Dream-org/Dream-VLA-7B - Hugging Face
https://huggingface.co/Dream-org/Dream-VLA-7B

34 35 36 39 Hugging Face says its new robotics model is so efficient it can run on a MacBook | TechCrunch
https://techcrunch.com/2025/06/04/hugging-face-says-its-new-robotics-model-is-so-efficient-it-can-run-on-a-macbook/

37 Train SmolVLA - phospho starter pack documentation
https://docs.phospho.ai/learn/train-smolvla

38 SmolVLA: Efficient Vision-Language-Action Model trained on ...
https://huggingface.co/blog/smolvla

40 Figure announces 'Helix,' an AI language model specialized for controlling humanoid robots at home - GIGAZINE
https://gigazine.net/gsc_news/en/20250223-figure-ai-helix/

41 42 43 44 45 46 47 Helix: A Vision-Language-Action Model for Generalist Humanoid Control
https://www.figure.ai/news/helix

48 50 52 53 54 Deep Dive into Robotics Learning Architectures
https://www.linkedin.com/pulse/deep-dive-robotics-learning-architectures-kai-xin-thia-ptq0f

49 51 55 56 57 NVIDIA announces 'Isaac GR00T N1', an open platform model for humanoid robots, useful for developing general-purpose robots that can operate in a variety of environments - GIGAZINE
https://gigazine.net/gsc_news/en/20250319-nvidia-isaac-gr00t-n1