

# Contents

<b>1 Chapter 6: Action-Interface Engineering -- Policies as Controllers</b>	<b>1</b>
1.1 Bridge: From Task Success to Movement Quality	1
1.2 WHY: Engineering Metrics for Learned Policies	2
1.2.1 6.1 The Gap Between Success and Deployability	2
1.2.2 6.2 What an Action Interface Is	2
1.2.3 6.3 The Controller Metric Bundle	2
1.2.4 6.4 Planning vs Control: A Diagnostic Decomposition	3
1.3 HOW: Action Wrappers and Classical Baselines	3
1.3.1 6.5 Action Scaling	3
1.3.2 6.6 Low-Pass Filtering (EMA)	3
1.3.3 6.7 Proportional Controller Baseline	4
1.4 BUILD IT: Action Wrappers and Metrics from Scratch (6.8)	4
1.4.1 6.8.1 Action Scaling Wrapper	4
1.4.2 6.8.2 Low-Pass Filter Wrapper	5
1.4.3 6.8.3 Proportional Controller	5
1.4.4 6.8.4 Controller Metrics	6
1.4.5 6.8.5 Unified Evaluation Loop	6
1.4.6 6.8.6 Verify Everything	6
1.5 WHAT: Experiments and Expected Results (Run It)	7
1.5.1 6.9 Quick Start: Full Pipeline	7
1.5.2 6.10 Experiment 1: Action Scaling Sweep	7
1.5.3 6.11 Experiment 2: Low-Pass Filter Sweep	8
1.5.4 6.12 Experiment 3: PD Controller Baseline	9
1.5.5 6.13 Experiment 4: Planning vs Control Decomposition	10
1.5.6 6.14 Engineering Metrics Dashboard	10
1.5.7 6.15 Video Recordings	10
1.6 Summary	11
1.6.1 Concepts Introduced in This Chapter	11
1.6.2 Files Generated	11
1.6.3 Artifacts	11
1.6.4 What Comes Next	11

## 1 Chapter 6: Action-Interface Engineering -- Policies as Controllers

**Week 6 Goal:** Treat learned policies like controllers and quantify stability with engineering metrics.

---

### 1.1 Bridge: From Task Success to Movement Quality

In Chapter 5, we trained SAC+HER on FetchPickAndPlace -- a task requiring six-phase coordination (approach, descend, grasp, lift, carry, place). We focused on whether the robot could *succeed*: reach the goal, grasp the object, place it correctly. But success rate alone does not tell us whether the robot's *behavior* is acceptable. A policy that achieves 100% success but jitters wildly, saturates actuators, or takes erratic paths would be unusable on real hardware.

This chapter shifts the question from "can the robot succeed?" to "how well does the robot behave?" We treat trained policies as **controllers** -- components in a feedback loop -- and evaluate them with the same metrics a control engineer would use: smoothness, peak effort,

time-to-success, and path efficiency. No new training is needed; we reuse checkpoints from Chapter 4 and study how the **action interface** (scaling, filtering) affects movement quality.

We also build a simple proportional controller as a **classical baseline**. Comparing RL against this baseline lets us separate two failure modes: **control problems** (tasks a simple controller can solve) from **planning problems** (tasks that require learned strategies).

## 1.2 WHY: Engineering Metrics for Learned Policies

### 1.2.1 6.1 The Gap Between Success and Deployability

Reinforcement learning optimizes a scalar reward signal. For sparse-reward tasks, that signal is binary:  $R = 0$  (success) or  $R = -1$  (failure). A policy that maximizes expected return under this signal has no reason to prefer smooth trajectories over jerky ones, short paths over wandering ones, or gentle actions over saturating ones -- as long as it eventually reaches the goal.

In robotics, this gap matters. Actuators have torque limits, joints have velocity constraints, and objects in contact can slip or break under abrupt force changes. A policy's **action interface** -- how raw neural network outputs map to physical commands -- determines whether learned behavior is physically plausible.

### 1.2.2 6.2 What an Action Interface Is

The action interface sits between the policy's output  $a_{\text{raw}} \in [-1, 1]^4$  and the environment's `step()` function. In the simplest case (the default), the interface is the identity:  $a_{\text{env}} = a_{\text{raw}}$ . But we can insert transformations at evaluation time without retraining:

1. **Action scaling:**  $a_{\text{env}} = \text{clip}(\alpha \cdot a_{\text{raw}}, [-1, 1])$ , where  $\alpha > 0$  controls the magnitude.
2. **Low-pass filtering:**  $a_{\text{env}}^{(t)} = \alpha \cdot a_{\text{raw}}^{(t)} + (1 - \alpha) \cdot a_{\text{env}}^{(t-1)}$ , where  $\alpha \in (0, 1]$  controls the smoothing strength.

These are **eval-time wrappers** -- they modify how actions are delivered without changing the learned policy weights. This lets us study the sensitivity of a trained policy to its action interface.

### 1.2.3 6.3 The Controller Metric Bundle

Instead of a single success rate, we evaluate policies on a bundle of engineering metrics. We find these capture different aspects of movement quality:

Metric	Formula	What It Measures
Success rate	$\frac{1}{N} \sum_{i=1}^N \mathbb{1}[\text{success}_i]$	Task completion
Time-to-success (TTS)	First step $t$ where $\ g_{\text{achieved}} - g_{\text{desired}}\  < \epsilon$	Speed
Smoothness	$\frac{1}{T-1} \sum_{t=1}^{T-1} \ a_t - a_{t-1}\ ^2$	Action jitter (lower = smoother)
Peak action	$\max_{t,d}  a_{t,d} $	Actuator saturation
Path length	$\sum_{t=1}^T \ p_t - p_{t-1}\ $	Gripper travel (meters)
Action energy	$\sum_{t=0}^{T-1} \ a_t\ ^2$	Total effort

Here  $a_t \in \mathbb{R}^4$  is the action at step  $t$ ,  $p_t \in \mathbb{R}^3$  is the gripper position,  $T$  is the episode length (50 steps for Fetch), and  $\epsilon = 0.05$  m is the success threshold.

#### 1.2.4 6.4 Planning vs Control: A Diagnostic Decomposition

Not all tasks are equally hard for the same reasons. We propose a simple decomposition:

1. **Evaluate RL** (SAC+HER, trained policy) on the task:  $SR_{RL}$
2. **Evaluate PD** (proportional controller, hand-tuned) on the same task:  $SR_{PD}$
3. **Compute the gap:**  $\Delta = SR_{RL} - SR_{PD}$

The interpretation depends on both the gap and the absolute performance. When  $\Delta \approx 0$  and both success rates are high, the task is a pure *control* problem -- a simple controller suffices, so the learned policy's advantage lies in movement quality rather than strategy. When  $\Delta \gg 0$ , the task requires *planning*: the RL policy has learned multi-phase behavior or contact sequencing that no fixed-gain controller can replicate. When  $\Delta \approx 0$  but both success rates are low, neither approach works, which suggests the task may be fundamentally harder or require a different formulation entirely.

This decomposition is not rigorous (it conflates many factors), but we find it practically useful for diagnosing where to focus debugging effort.

### 1.3 HOW: Action Wrappers and Classical Baselines

#### 1.3.1 6.5 Action Scaling

**Definition (Action Scaling).** Given a scale factor  $\alpha > 0$  and a raw action  $a_{raw} \in [-1, 1]^d$ , the scaled action is:

$$a_{scaled} = \text{clip}(\alpha \cdot a_{raw}, a_{low}, a_{high})$$

where  $a_{low}, a_{high}$  are the environment's action space bounds (both  $[-1, 1]^4$  for Fetch).

**Intuition:** Scaling down ( $\alpha < 1$ ) makes the policy take smaller steps -- finer control at the cost of speed -- while scaling up ( $\alpha > 1$ ) amplifies actions, producing faster movement but more saturation and jitter.

**Prediction:** Smoothness should increase monotonically with  $\alpha$ , since larger actions produce larger differences between consecutive steps. Success rate may degrade at extreme scales where either the actions are too small to accomplish the task or too large to maintain precision.

#### 1.3.2 6.6 Low-Pass Filtering (EMA)

**Definition (Exponential Moving Average Filter).** Given a filter coefficient  $\alpha \in (0, 1]$ , the filtered action at step  $t$  is:

$$a_{out}^{(t)} = \alpha \cdot a_{raw}^{(t)} + (1 - \alpha) \cdot a_{out}^{(t-1)}$$

with  $a_{out}^{(0)} = a_{raw}^{(0)}$  (no prior history).

**Intuition:** The filter blends the current raw action with the previous output. When  $\alpha = 1$ , there is no filtering (pass-through). When  $\alpha \rightarrow 0$ , the output barely changes -- extremely smooth but unresponsive.

**Prediction:** Smoothness should decrease as  $\alpha$  decreases, since heavier filtering averages out consecutive action differences. However, for tasks requiring fast reactions (such as switching behavior at the moment of contact during pushing), heavy filtering may delay the transition and hurt success rate.

### 1.3.3 6.7 Proportional Controller Baseline

**Definition (Proportional Controller).** For a Fetch environment with gripper position  $p \in \mathbb{R}^3$  and desired goal  $g \in \mathbb{R}^3$ , the proportional controller outputs:

$$a_{xyz} = K_p \cdot (g - p), \quad a_{gripper} = 0$$

where  $K_p > 0$  is the proportional gain. The full action  $a = [a_{xyz}, a_{gripper}] \in \mathbb{R}^4$  is clipped to  $[-1, 1]^4$ .

**For FetchPush**, the controller uses a two-phase strategy:

1. **Approach:** If the gripper is far from the object ( $\|p - p_{obj}\| > 0.05$  m), move toward the object.
2. **Push:** Once near the object, push it toward the goal.

**Note:** MuJoCo's Fetch robot already has internal PD controllers on each joint, so our Cartesian proportional controller operates on top of those -- it computes a desired Cartesian displacement, and MuJoCo's internal controllers handle the joint-level tracking.

**Why this matters:** The PD controller gives us a calibration point. If it solves the task, the learned policy's value lies in movement quality rather than strategy; if it fails, we know the task requires planning that a simple controller cannot provide.

## 1.4 BUILD IT: Action Wrappers and Metrics from Scratch (6.8)

This section walks through the pedagogical implementations in `scripts/labs/action_interface.py`. Each component maps directly to the definitions above.

### 1.4.1 6.8.1 Action Scaling Wrapper

The scaling wrapper multiplies raw actions by a constant factor, then clips to the action space bounds:

```
--8-- "scripts/labs/action_interface.py:action_scaling_wrapper"
```

```
!!! lab "Checkpoint" Verify scaling math with a quick test: "python import gymnasium import
gymnasium_robotics # noqa import numpy as np
```

```
env = gymnasium.make("FetchReach-v4")
from scripts.labs.action_interface import ActionScalingWrapper
```

```
wrapped = ActionScalingWrapper(env, scale=0.5)
# Action space bounds should be unchanged
assert wrapped.action_space.shape == env.action_space.shape
```

```
# scale=0.5 should halve actions: [1,1,1,1] -> [0.5,0.5,0.5,0.5]
result = wrapped.action(np.array([1.0, 1.0, 1.0, 1.0]))
assert np.allclose(result, [0.5, 0.5, 0.5, 0.5])
```

```
# scale=2.0 should clip: [0.8,0.8,0.8,0.8] -> [1.0,1.0,1.0,1.0]
wrapped2 = ActionScalingWrapper(env, scale=2.0)
result2 = wrapped2.action(np.array([0.8, 0.8, 0.8, 0.8]))
assert np.allclose(result2, [1.0, 1.0, 1.0, 1.0])

print("ActionScalingWrapper: PASS")
\`\`\`
```

### 1.4.2 6.8.2 Low-Pass Filter Wrapper

The EMA filter smooths consecutive actions, reducing jitter at the cost of responsiveness:

```
--8<-- "scripts/labs/action_interface.py:lowpass_filter_wrapper"
```

The key design choice: the filter state resets on `env.reset()`. Without this, the first action of a new episode would be blended with the last action of the previous episode -- a subtle bug that would affect reproducibility.

!!! lab "Checkpoint" Verify the EMA formula: `python from scripts.labs.action_interface import LowPassFilterWrapper import gymnasium import gymnasium_robotics # noqa import numpy as np`

```
env = gymnasium.make("FetchReach-v4")

# alpha=1.0 should be identity (pass-through)
filt = LowPassFilterWrapper(env, alpha=1.0)
filt.reset()
a = np.array([0.5, -0.3, 0.1, 0.0])
assert np.allclose(filt.action(a), a), "alpha=1.0 must be identity"

# alpha=0.5: first action passes through, second blends
filt2 = LowPassFilterWrapper(env, alpha=0.5)
filt2.reset()
a1 = np.array([0.5, 0.5, 0.5, 0.5])
out1 = filt2.action(a1) # First action: no prior -> pass-through
a2 = np.array([-0.5, -0.5, -0.5, -0.5])
out2 = filt2.action(a2) # Second: 0.5*(-0.5) + 0.5*(0.5) = 0.0
assert np.allclose(out2, [0.0, 0.0, 0.0, 0.0])

print("LowPassFilterWrapper: PASS")
\`\`\`
```

### 1.4.3 6.8.3 Proportional Controller

The P-controller implements the same `predict(obs, deterministic)` interface as an SB3 model, so our evaluation loop can treat RL policies and classical controllers interchangeably:

```
--8<-- "scripts/labs/action_interface.py:proportional_controller"
```

!!! lab "Checkpoint" Verify convergence on FetchReach-v4: `python from scripts.labs.action_interface import ProportionalController import gymnasium import gymnasium_robotics # noqa`

```
env = gymnasium.make("FetchReach-v4")
controller = ProportionalController("FetchReach-v4", kp=10.0)

obs, info = env.reset(seed=42)
```

```

for step in range(50):
    action, _ = controller.predict(obs, deterministic=True)
    obs, reward, term, trunc, info = env.step(action)
    if info.get("is_success", False):
        print(f"ProportionalController: PASS (converged in {step+1} steps)")
        break
else:
    print("ProportionalController: FAIL (did not converge in 50 steps)")
env.close()

```

Expected: convergence within 5-10 steps for Reach with  $K_p = 10$ .

#### 1.4.4 6.8.4 Controller Metrics

The metric dataclass bundles all engineering measurements for a single episode, while the computation function extracts them from trajectory data:

```
--8<-- "scripts/labs/action_interface.py:controller_metrics"
```

The smoothness metric deserves attention. We define it as the *mean squared action difference*:

$$\text{smoothness} = \frac{1}{T-1} \sum_{t=1}^{T-1} \|a_t - a_{t-1}\|^2$$

This is zero when actions are constant (perfectly smooth) and large when actions oscillate. We find this more informative than raw action variance because it captures *consecutive* changes -- a policy that alternates between +1 and -1 every step has high smoothness cost even if its action mean is zero.

#### 1.4.5 6.8.5 Unified Evaluation Loop

The `run_controller_eval()` function runs N episodes with any policy-like object -- applying optional wrappers along the way -- and returns both aggregate statistics and per-episode details:

```
--8<-- "scripts/labs/action_interface.py:run_controller_eval"
```

The wrapper composition pattern (line 309-310 in the source) applies wrappers in list order with the last wrapper being outermost, following the standard Gymnasium convention. This means scaling and filtering can be stacked so that the raw action is first scaled, then smoothed:

```

wrappers = [
    (ActionScalingWrapper, {"scale": 0.5}),
    (LowPassFilterWrapper, {"alpha": 0.8}),
]

```

#### 1.4.6 6.8.6 Verify Everything

Run the built-in verification suite (~30 seconds):

```
bash docker/dev.sh python scripts/labs/action_interface.py --verify
```

Expected output:

```

=== verify_action_scaling ===
Space preserved: PASS

```

```

scale=0.5 math: PASS
scale=2.0 clipping: PASS
scale=1.0 identity: PASS

=== verify_lowpass_filter ===
alpha=1.0 identity: PASS
alpha=0.5 smoothing: PASS
Reset clears state: PASS

=== verify_proportional_controller ===
PD converged in N steps: PASS

=== verify_metrics_finite ===
All metrics finite: PASS

=== verify_run_controller_eval ===
success_rate >= 80%: PASS
All aggregate metrics finite: PASS

All 5 checks passed.

```

---

## 1.5 WHAT: Experiments and Expected Results (Run It)

All experiments use checkpoints from Chapter 4 (SAC+HER, FetchReach-v4 and FetchPush-v4). No new training is needed -- this is an evaluation-only chapter.

### 1.5.1 6.9 Quick Start: Full Pipeline

```

# Full pipeline: scaling + filter + baseline + compare + video (both envs)
bash docker/dev.sh python scripts/ch06_action_interface.py all --seed 0 --include-push

```

This runs approximately 2,500 episodes across all conditions and takes about 5-8 minutes on a DGX with GPU. Individual subcommands are described below.

### 1.5.2 6.10 Experiment 1: Action Scaling Sweep

```

bash docker/dev.sh python scripts/ch06_action_interface.py scaling --seed 0 --include-push

```

We evaluate the SAC+HER policy at scale factors  $\{0.25, 0.5, 0.75, 1.0, 1.5, 2.0\}$ , running 100 episodes per condition.

#### FetchReach-v4 -- Scaling Results:

Scale	Success	Smoothness	Peak  a	Path (m)	Energy	TTS
0.25	100%	0.0050	0.982	0.144	27.0	9.1
0.50	100%	0.0110	0.982	0.145	17.3	4.8
0.75	100%	0.0185	0.982	0.147	14.0	3.4
<b>1.00</b>	<b>100%</b>	<b>0.0284</b>	<b>0.982</b>	<b>0.149</b>	<b>12.4</b>	<b>2.7</b>
1.50	100%	0.0511	0.982	0.170	12.3	2.7
2.00	100%	0.9476	0.982	1.025	23.6	2.7

Smoothness monotonicity: **PASS** (increases with scale as predicted).

## Action Scaling -- FetchReach

**Interpretation:** FetchReach is robust to all scale factors -- 100% success everywhere -- but the movement quality changes dramatically. At scale=0.25, the gripper crawls (TTS=9.1 steps) yet is whisper-smooth (0.005), whereas at scale=2.0 it arrives instantly (TTS=2.7) but smoothness explodes to 0.95 because the clipping at  $[-1, 1]$  causes the policy to bang between saturation limits. The energy U-shape (27 at 0.25, minimum 12.3 at 1.5, back to 23.6 at 2.0) confirms that the *trained* scale of 1.0 sits near the energy-efficient sweet spot, which makes sense given that SAC's entropy regularization implicitly discourages unnecessary action magnitude.

### FetchPush-v4 -- Scaling Results:

Scale	Success	Smoothness	Peak  a	Path (m)	Energy	TTS
0.25	25%	0.073	0.961	0.391	58.1	29.0
0.50	66%	0.290	0.967	0.666	46.4	24.7
0.75	95%	0.431	0.969	0.870	39.8	18.7
<b>1.00</b>	<b>100%</b>	<b>0.522</b>	<b>0.970</b>	<b>1.005</b>	<b>34.2</b>	<b>14.7</b>
1.50	99%	0.730	0.972	1.322	37.9	13.7
2.00	98%	0.751	0.975	1.460	37.1	13.7

Smoothness monotonicity: **PASS**.

## Action Scaling -- FetchPush

**Interpretation:** Push shows a clear *minimum force threshold*. At scale=0.25, the gripper cannot push the block hard enough -- only 25% of episodes succeed (likely those where the block starts near the goal) -- and the S-curve from 25% to 100% reveals that pushing requires sustained contact forces that small actions cannot produce. Above scale=1.0, success drops slightly (99%, 98%) as oversized actions cause overshoot at contact.

## Action Scaling -- Combined

The combined plot contrasts Reach (flat at 100%) with Push (S-curve), providing our first evidence that Push is *structurally harder* because it has a force-threshold requirement that Reach does not.

### 1.5.3 6.11 Experiment 2: Low-Pass Filter Sweep

```
bash docker/dev.sh python scripts/ch06_action_interface.py filter --seed 0 --include-push
```

We evaluate at filter coefficients  $\alpha \in \{0.2, 0.4, 0.6, 0.8, 1.0\}$ .

### FetchReach-v4 -- Filter Results:

$\alpha$	Success	Smoothness	Peak  a	Path (m)	Energy	TTS
0.20	100%	0.0710	0.986	0.347	23.2	3.8
0.40	100%	0.0479	0.982	0.218	14.2	2.8
0.60	100%	0.0374	0.982	0.175	12.7	2.7
0.80	100%	0.0316	0.982	0.155	12.5	2.7
<b>1.00</b>	<b>100%</b>	<b>0.0284</b>	<b>0.982</b>	<b>0.149</b>	<b>12.4</b>	<b>2.7</b>

## Low-Pass Filter -- FetchReach

**Interpretation:** For Reach, the filter does not hurt success rate at all, but note an interesting effect: at  $\alpha = 0.2$  (heavy smoothing), smoothness actually *increases* to 0.071 (from 0.028 unfiltered), which seems counterintuitive since one might expect filtering to reduce smoothness.

The explanation is that smoothness measures *action differences*, and heavy filtering causes a persistent *lag* where the filtered output slowly chases the raw policy output, creating a non-zero difference at every step. Because the raw policy (no filter) happens to output fairly similar actions in consecutive steps for Reach, the filter adds more lag-induced differences than it removes in jitter. This is a good reminder that smoothness is not the same as “looks smooth to a human” -- it is a specific mathematical quantity that can increase even when the trajectory appears visually gentle.

#### FetchPush-v4 -- Filter Results:

$\alpha$	Success	Smoothness	Peak  a	Path (m)	Energy	TTS
0.20	85%	0.258	0.976	1.042	62.1	29.1
0.40	97%	0.305	0.973	0.936	43.4	20.5
0.60	99%	0.356	0.970	0.904	36.3	16.3
0.80	99%	0.440	0.969	0.949	33.9	15.3
<b>1.00</b>	<b>100%</b>	<b>0.522</b>	<b>0.970</b>	<b>1.005</b>	<b>34.2</b>	<b>14.7</b>

Low-Pass Filter -- FetchPush

**Interpretation:** Here the filter *does* cost success rate -- at  $\alpha = 0.2$ , success drops to 85% because Push requires temporally precise control where the policy must switch from “approach” to “push” behavior at the moment of contact. Heavy filtering delays this transition, causing the gripper to overshoot or slide past the block, so the 15% success drop at  $\alpha = 0.2$  is direct evidence that Push's learned strategy depends on rapid action changes that filtering suppresses.

#### 1.5.4 6.12 Experiment 3: PD Controller Baseline

`bash docker/dev.sh python scripts/ch06_action_interface.py baseline --include-push`

We evaluate the proportional controller at  $K_p \in \{5, 10, 20\}$ .

#### FetchReach-v4 -- PD Baseline:

$K_p$	Success	Smoothness	Peak  a	Path (m)	Energy	TTS
5.0	100%	0.0010	0.552	0.139	1.82	6.0
10.0	100%	0.0060	0.918	0.140	3.80	3.1
20.0	100%	0.0185	0.985	0.144	6.00	2.6

The PD controller achieves 100% on Reach at all gains, and notably at  $K_p = 5$  it is *smoother than the RL policy* (0.001 vs 0.028) while using far less energy (1.82 vs 12.44). The price is that it takes longer (TTS=6.0 vs 2.7), which illustrates the speed-versus-smoothness tradeoff that a controls engineer would weigh in deployment.

#### FetchPush-v4 -- PD Baseline:

$K_p$	Success	Smoothness	Peak  a	Path (m)	Energy	TTS
5.0	5%	0.687	0.869	0.699	14.6	1.0
10.0	5%	2.255	1.000	1.258	42.7	1.0
20.0	5%	2.099	1.000	1.107	52.9	1.0

The PD controller **fails catastrophically on Push** -- only 5% success (essentially chance) -- because a proportional controller moves the gripper toward the goal but cannot orchestrate

the approach-then-push sequence. It does not know to first move to the *block* and then push the block toward the *goal*, which provides the clearest evidence that Push requires planning.

### 1.5.5 6.13 Experiment 4: Planning vs Control Decomposition

```
bash docker/dev.sh python scripts/ch06_action_interface.py compare --include-push
```

Planning vs Control Decomposition

Environment	RL (SAC+HER)	PD (best $K_p$ )	Gap	Interpretation
FetchReach-v4	100%	100%	0%	Pure <b>control</b> task
FetchPush-v4	100%	5%	+95%	Requires <b>planning</b>

**FetchReach** is a pure control task -- both RL and PD solve it perfectly, so the learned policy's advantage over PD lies not in *whether* it succeeds but in *how fast* it succeeds (TTS=2.7 vs 6.0 at  $K_p = 5$ ), a tradeoff that a controls engineer would weigh against smoothness in deployment.

**FetchPush** requires planning. The 95-percentage-point gap proves that the RL policy has learned strategies -- approach, contact, push -- that no fixed-gain controller can replicate, which means that when debugging a Push failure, the decomposition tells us to look at the *strategy* (reward shaping, exploration) rather than the action interface.

### 1.5.6 6.14 Engineering Metrics Dashboard

The following triptychs show how three engineering metrics vary with action scale for each environment:

Engineering Metrics -- FetchReach

Engineering Metrics -- FetchPush

**Key observations:** For both environments, TTS drops quickly as scale increases but plateaus beyond scale=1.0, which means bigger actions do not make Push faster once the force threshold is met. Path length, by contrast, increases monotonically with scale because overshoot adds travel distance -- at scale=2.0, the Reach gripper travels 1.03 m (7x the 0.14 m at scale=1.0) due to oscillation. Perhaps most striking is the energy U-shape: the effort cost is minimized near scale=1.0 -- exactly where the policy was trained -- since SAC's entropy regularization implicitly penalizes unnecessary action magnitude, producing an energy-efficient operating point.

### 1.5.7 6.15 Video Recordings

The pipeline also records side-by-side comparison videos:

*# Videos are saved to videos/ directory*

```
bash docker/dev.sh python scripts/ch06_action_interface.py video --seed 0 --include-push
```

Video	File	What to Watch
RL vs PD (Reach)	videos/ch06_fetchreachv4_rl_vs_pd.mp4	Both succeed; PD is smoother, RL is faster
RL vs PD (Push)	videos/ch06_fetchpushv4_rl_vs_pd.mp4	RL pushes to goal; PD flails
Scaling (Reach)	videos/ch06_fetchreachv4_scaling.mp4	0.25x crawls, 1x normal, 2x jitters
Scaling (Push)	videos/ch06_fetchpushv4_scaling.mp4	0.25x fails, 1x succeeds, 2x overshoots
Filter (Reach)	videos/ch06_fetchreachv4_filter.mp4	All succeed; alpha=0.2 has visible lag
Filter (Push)	videos/ch06_fetchpushv4_filter.mp4	alpha=0.2 misses; alpha=1.0 succeeds

Video	File	What to Watch
-------	------	---------------

## 1.6 Summary

### 1.6.1 Concepts Introduced in This Chapter

Concept	Definition
Action interface	Transformation between policy output and environment input
Action scaling	Multiply actions by constant factor, clip to bounds
Low-pass filter (EMA)	Smooth consecutive actions: $a_{\text{out}}^{(t)} = \alpha a_{\text{raw}}^{(t)} + (1 - \alpha) a_{\text{out}}^{(t-1)}$
Smoothness	Mean squared action difference: $\frac{1}{T-1} \sum \ a_t - a_{t-1}\ ^2$
Time-to-success (TTS)	First step achieving goal within $\epsilon$
Peak action	Maximum absolute action component (saturation indicator)
Path length	Total gripper travel distance in meters
Action energy	Sum of squared action magnitudes (effort proxy)
Controller metric bundle	Collection of engineering metrics evaluated per episode
Proportional controller	$a = K_p(g - p)$ : error-driven Cartesian control
Planning-vs-control decomposition	Compare RL success to PD baseline to classify task type

### 1.6.2 Files Generated

File	Purpose
scripts/labs/action_interface.py	Pedagogical wrappers and metrics (Build It)
scripts/ch06_action_interface.py	Chapter script: scaling, filter, baseline, compare, video
scripts/ch06_plot.py	Figure generation from JSON results
tutorials/ch06_action_interface.md	This tutorial

### 1.6.3 Artifacts

Artifact	Location
Scaling results	results/ch06_scaling_{env}_seed0.json
Filter results	results/ch06_filter_{env}_seed0.json
Baseline results	results/ch06_baseline_{env}.json
Comparison	results/ch06_comparison.json
Figures	figures/ch06_*.png (8 figures)
Videos	videos/ch06_*.mp4 (6 videos)

### 1.6.4 What Comes Next

Chapter 7 addresses **robustness** -- how policies degrade under noise, domain randomization, and observation perturbations. Where this chapter asked "how well does the robot behave under ideal conditions?", Chapter 7 asks "how well does it behave when conditions change?", and the action wrappers and engineering metrics we built here will serve as the evaluation framework for that robustness testing.