## Contents

# 1 Chapter 7: Robustness Curves -- Quantifying Policy Brittleness

**Week 7 Goal:** Produce degradation curves -- success rate as a function of noise -- with confidence intervals, turning qualitative robustness claims into quantitative brittleness fingerprints.

---

## 1.1 Bridge: From Movement Quality to Noise Tolerance

In Chapter 6, we treated trained policies as controllers and evaluated them with engineering metrics: smoothness, peak effort, time-to-success, path efficiency. Those experiments assumed *ideal conditions* -- the policy received perfect observations and its actions were executed exactly as commanded. Under these conditions, SAC+HER achieved 100% success on both FetchReach and FetchPush.

But real hardware has neither perfect sensors nor perfect actuators. Joint encoders have quantization noise, force-torque sensors drift, cameras introduce pixel noise, and motors have backlash and friction that distort commanded actions. A policy that works flawlessly in simulation may fail catastrophically on a physical robot -- not because the policy is ”wrong,” but because it was never tested under the conditions it would actually face.

This chapter introduces **parametric noise sweeps**: we systematically inject Gaussian noise at increasing levels and measure how success rate degrades. The result is a **degradation curve** -- a ”brittleness fingerprint” that tells us *how much* noise a policy can tolerate before it fails, and *how quickly* it degrades. We also test whether the low-pass filter from Chapter 6 can mitigate action noise. In Chapter 8, we will take the next step: testing whether the brittleness patterns we find here transfer to a different simulator.

---

## 1.2 WHY: The Robustness Problem

### 1.2.1 7.1 The Gap Between Simulation and Deployment

A policy trained in simulation encounters two types of imprecision when deployed on hardware:

1. **Sensor noise** (observation): The state estimate $\hat{o}_t$ differs from the true state $o_t$ due to measurement error. Joint encoders have finite resolution, cameras introduce pixel noise, and state estimators (e.g., Kalman filters) have residual error.
2. **Actuator imprecision** (action): The executed action $\hat{a}_t$ differs from the commanded action $a_t$ due to backlash, friction, motor deadbands, and transmission delay.

In simulation, both noise sources are zero by default. This means a 100% success rate in simulation tells us nothing about deployment robustness -- it is a necessary but not sufficient condition. The question this chapter addresses is: **how does success rate degrade as we introduce controlled amounts of noise?**

### 1.2.2 7.2 Noise Models: Observation and Action

We model both noise sources as additive Gaussian perturbations, which we find is a reasonable first approximation for many real-world noise sources (by the central limit theorem, the sum of many small independent errors tends toward Gaussian).

**Definition (Observation Noise).** Given the true observation $o_t$ and a noise standard deviation $\sigma_{\text{obs}} \geq 0$, the noisy observation is:

$$\tilde{o}_t = o_t + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma_{\text{obs}}^2 I)$$

where $I$ is the identity matrix with dimension matching $o_t$. We apply noise only to the `observation` key of the Gymnasium dict -- the `desired_goal` and `achieved_goal` keys are left untouched because they represent ground-truth target positions, not sensor readings.

**Non-example:** Observation noise does *not* model systematic calibration errors (constant bias). A miscalibrated sensor that always reads 2 cm too high would shift the mean of $\tilde{o}_t$, not just add variance. Our Gaussian model captures random fluctuations -- the kind that change from timestep to timestep -- not persistent offsets. Testing bias robustness would require a different noise model (additive constant rather than zero-mean Gaussian).

**Definition (Action Noise).** Given the policy's commanded action $a_t$ and a noise standard deviation $\sigma_{\text{act}} \geq 0$, the executed action is:

$$\tilde{a}_t = \text{clip}(a_t + \eta,\ a_{\text{low}},\ a_{\text{high}}), \quad \eta \sim \mathcal{N}(0, \sigma^2_{\text{act}} I)$$

The clipping to $[a_{\text{low}}, a_{\text{high}}]$ = $[-1, 1]^4$ ensures the environment never receives out-of-range commands.

**Non-example:** Action noise does *not* model communication delay (latency). A motor that receives commands 50 ms late executes the *correct* action at the *wrong* time -- a temporal shift, not a per-step random perturbation. Our model captures instantaneous execution error (backlash, friction) where the right command is sent at the right time but executed imprecisely.

**Design choice:** We inject noise on the `observation` key only, not on goals. This models the physical reality: the robot's proprioceptive sensors (joint angles, velocities) are noisy, but the goal position comes from a separate planning system (e.g., a pick-and-place planner specifying where to put the object). If goals were also noisy, we would be testing a different failure mode -- goal estimation error -- which we choose to study separately.

**Typical sigma ranges:**

| Noise type | $\sigma$ range | Physical interpretation |
|---|---|---|
| Observation | 0.0 -- 0.2 | Joint encoder noise (rad), position error (m) |
| Action | 0.0 -- 0.5 | Motor backlash, transmission error |

For the Fetch observation vector (10D for Reach, 25D for Push), a noise std of 0.01 corresponds to roughly 1 cm of position error or 0.01 rad of joint angle noise -- a realistic level for consumer-grade hardware. A noise std of 0.1 is extreme -- 10 cm of position error -- and we extend up to $\sigma = 0.2$ to find where even robust policies break down. For action noise, $\sigma = 0.5$ represents half the full action range $[-1, 1]$ -- severe actuator imprecision that should challenge any controller.

### 1.2.3 7.3 Degradation Curves: Success Rate Under Noise

The core idea of this chapter is simple: for a fixed policy, sweep $\sigma$ from 0 to some maximum and plot success rate:

**Definition (Degradation Curve).** For a policy $\pi$, environment $E$, noise type $\nu \in \{\text{obs}, \text{act}\}$, and noise levels $\Sigma = \{\sigma_0, \sigma_1, \dots, \sigma_K\}$ with $\sigma_0 = 0$:

$$\text{SR}(\sigma) = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}[\text{success}_i(\pi, E, \nu, \sigma)]$$

where $N$ is the number of evaluation episodes per noise level. The degradation curve is the function $\sigma \mapsto \text{SR}(\sigma)$.

A single success rate (the $\sigma = 0$ case) is a special case of this curve. The full curve is strictly more informative: it reveals *how quickly* performance degrades, *where* the critical threshold lies, and *whether* degradation is gradual or cliff-like.

**Cross-seed aggregation:** Because RL policies vary across training seeds (Henderson et al., 2018), a single-seed degradation curve may be misleading. When multiple seeds are available, we aggregate:

$$\overline{\mathsf{SR}}(\sigma) = \frac{1}{S}\sum_{s=1}^{S} \mathsf{SR}_s(\sigma), \quad \mathsf{CI}_{95\%} = \overline{\mathsf{SR}} \pm z \cdot \frac{s_{\mathsf{SR}}}{\sqrt{S}}$$

where $S$ is the number of seeds, $s_{\mathsf{SR}}$ is the standard deviation across seeds, and $z = 1.96$ for large samples (or $z \approx 2.0$ for $S < 30$ as a conservative approximation).

#### 1.2.4 7.4 Three Summary Metrics

A degradation curve contains rich information, but for comparison across environments and noise types, we compress it into three scalar metrics:

**Definition (Critical Sigma).** The first noise level where mean success rate drops below 50%:

$$\sigma^* = \min\{\sigma \in \Sigma : \overline{\mathsf{SR}}(\sigma) < 0.50\}$$

If $\overline{\mathsf{SR}}$ never drops below 50%, we report $\sigma^* =$ None (the policy is robust across the entire tested range). Higher $\sigma^*$ means more robust.

**Non-example:** $\sigma^* =$ None does *not* always mean "robust." A policy that starts at 5% success and stays at 5% under all noise levels never crosses the 50% threshold, so $\sigma^* =$ None -- but it is uniformly *incapable*, not robustly *capable*. We will see this exact case with the PD controller on FetchPush (Section 7.12). Critical sigma is only meaningful when baseline performance is high.

**Definition (Degradation Slope).** The slope of a linear regression of success rate versus sigma:

$$\overline{\mathsf{SR}}(\sigma) \approx m \cdot \sigma + b$$

More negative $m$ means faster degradation -- a more brittle policy. A slope near zero means the policy degrades slowly (or not at all) across the tested range.

**Non-example:** A slope of zero does *not* always mean the policy is robust. It can also mean the policy is already failing uniformly -- a flat line at 5% success has slope $m \approx 0$, but this tells us nothing about noise tolerance. The slope is most informative when the baseline SR is high and the curve actually has room to fall.

**Definition (Robustness AUC).** The area under the SR-vs-sigma curve, computed by trapezoidal integration:

$$\mathsf{AUC} = \sum_{k=1}^{K} \frac{1}{2}(\mathsf{SR}_{k-1} + \mathsf{SR}_k)(\sigma_k - \sigma_{k-1})$$

Higher AUC means more robust overall. This metric integrates behavior across all noise levels, whereas critical sigma focuses on the 50% threshold.

**Grounding example:** Consider three noise levels with success rates SR $= [1.0, 0.8, 0.4]$ at $\sigma = [0.0, 0.05, 0.1]$:

- Critical sigma: $\sigma^* = 0.1$ (first level where SR < 0.5)
- AUC: $0.5 \times (1.0 + 0.8) \times 0.05 + 0.5 \times (0.8 + 0.4) \times 0.05 = 0.045 + 0.030 = 0.075$
- Slope: linear regression gives $m \approx -6.0$ (in units of SR-per-unit-$\sigma$: each 0.01 increase in $\sigma$ costs about 0.06 in SR)

**Non-example:** A high robustness AUC does *not* mean the policy is good in absolute terms. A policy with 60% baseline success that never degrades under noise has an AUC of $0.06$ (for $\sigma \in [0, 0.1]$) -- higher than a 100% baseline policy that collapses to 0% at $\sigma = 0.1$, which has an AUC of only $0.05$. The 60% flat policy is more *noise-tolerant* despite worse baseline performance. AUC measures *noise tolerance*, not *overall quality* -- the baseline SR matters separately.

---

## 1.3 HOW: Measuring and Mitigating Brittleness

### 1.3.1 7.5 Noise Injection as an Eval-Time Wrapper

A key insight is that **noise injection does not require retraining** -- we wrap the environment with a `NoisyEvalWrapper` that adds Gaussian noise to observations or actions at evaluation time, leaving policy weights unchanged so that we are measuring the *existing* policy's sensitivity rather than training a new one.

This wrapper composition pattern follows the Gymnasium convention:

```
policy -> LowPassFilterWrapper (optional) -> NoisyEvalWrapper -> base env
```

The wrapper satisfies three properties that make it safe for systematic evaluation. First, it acts as the identity when both noise standard deviations are zero, so that an unperturbed sweep point recovers the exact baseline result (verified in the lab module's test suite). Second, it preserves goals: only the `observation` key is perturbed, while `desired_goal` and `achieved_goal` pass through untouched, which means the reward computation stays correct even under observation noise. Third, each wrapper instance carries its own numpy random generator, seeded independently from the environment seed, so that the same wrapper seed produces the same noise sequence regardless of the environment's internal state -- a requirement for reproducible degradation curves.

### 1.3.2 7.6 Cross-Seed Aggregation

RL training is stochastic, so different random seeds produce different policy weights -- and these policies may have different robustness profiles, since a policy that happens to learn a conservative strategy may be more noise-tolerant than one that learns an aggressive strategy even if both achieve 100% success under zero noise. Henderson et al. (2018) showed that single-seed RL results are unreliable, which means a degradation curve from one seed may not represent the algorithm's typical behavior. Our aggregation formula addresses this by computing mean, standard deviation, and 95% confidence intervals across seeds at each noise level, using a conservative $z \approx 2.0$ (instead of the normal approximation $z = 1.96$) for small sample sizes where $S < 30$.

### 1.3.3 7.7 Mitigation via Low-Pass Filtering

In Chapter 6, we introduced the `LowPassFilterWrapper` -- an exponential moving average filter that smooths consecutive actions:

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

where $\alpha \in (0, 1]$ controls the smoothing strength ($\alpha = 1$ is no filtering).

**Hypothesis:** Low-pass filtering should help mitigate *action* noise (by smoothing out random perturbations) but may hurt tasks requiring fast transitions (as we saw in Ch06, where Push's success dropped from 100% to 85% at $\alpha = 0.2$).

**Wrapper stacking for mitigation:**

`policy -> LowPassFilterWrapper -> NoisyEvalWrapper -> base env`

The ordering matters: the LPF smooths the policy's *intended* actions, and then the NoisyEvalWrapper adds noise to the smoothed actions. This models the physical situation where a software-side filter cleans up commands before they reach noisy actuators.

We sweep $\alpha$ values $\{0.3, 0.5, 0.7, 1.0\}$ at a fixed action noise of $\sigma_{\text{act}} = 0.5$ -- chosen because our extended noise range now reaches high enough to actually stress the policy, giving the filter something meaningful to mitigate.

---

## 1.4 BUILD IT: Robustness Toolkit from Scratch (7.8)

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

### 1.4.1 7.8.1 NoisyEvalWrapper

The wrapper implements the noise models from Section 7.2. For observations: $\tilde{o}_t = o_t + \epsilon$, $\epsilon \sim \mathcal{N}(0, \sigma^2_{\text{obs}} I)$. For actions: $\tilde{a}_t = \text{clip}(a_t + \eta)$, $\eta \sim \mathcal{N}(0, \sigma^2_{\text{act}} I)$.

The three methods that do the actual noise injection -- reset, step, and _add_obs_noise -- map directly to these formulas:

`--8<-- "scripts/labs/robustness.py:noisy_eval_wrapper_core"`

??? lab ”Full NoisyEvalWrapper class (click to expand)” The complete class includes the constructor (storing noise stds and creating an independent RNG) and the docstring: python `--8<-- "scripts/labs/robustness.py:noisy_eval_wrapper"`

!!! lab ”Checkpoint” Verify the wrapper's three key properties: “‘python import gymnasium import gymnasium_robotics # noqa import numpy as np

```
from scripts.labs.robustness import NoisyEvalWrapper

# 1. Zero noise = identity
env_zero = NoisyEvalWrapper(gymnasium.make("FetchReach-v4"),
                             obs_noise_std=0.0, act_noise_std=0.0)
obs_z, _ = env_zero.reset(seed=42)
env_ref = gymnasium.make("FetchReach-v4")
obs_r, _ = env_ref.reset(seed=42)
assert np.allclose(obs_z["observation"], obs_r["observation"])
print("Zero noise = identity: PASS")

# 2. Goals preserved under obs noise
env_noisy = NoisyEvalWrapper(gymnasium.make("FetchReach-v4"),
                              obs_noise_std=0.1)
obs_n, _ = env_noisy.reset(seed=42)
assert np.allclose(obs_n["desired_goal"], obs_r["desired_goal"])
assert not np.allclose(obs_n["observation"], obs_r["observation"])
print("Goals preserved, observation changed: PASS")

# 3. Action noise clips to bounds
env_act = NoisyEvalWrapper(gymnasium.make("FetchReach-v4"),
                            act_noise_std=10.0)
```

```
env_act.reset(seed=42)
obs_a, _, _, _, _ = env_act.step(np.ones(4, dtype=np.float32))
print("Action noise with large sigma: env stable: PASS")
```

### 1.4.2 7.8.2 NoiseSweepResult

The result dataclass stores all metrics from evaluating at one noise level:

```
--8<-- "scripts/labs/robustness.py:noise_sweep_result"
```

!!! lab "Checkpoint" Verify the round-trip serialization: "'python from scripts.labs.robustness import NoiseSweepResult

```
r = NoiseSweepResult(noise_type="obs", noise_std=0.05,
                     success_rate=0.85, return_mean=-12.0,
                     episode_successes=[True]*17 + [False]*3)
d = r.to_dict()
assert d["noise_type"] == "obs"
assert d["noise_std"] == 0.05
assert d["success_rate"] == 0.85
assert len(d["episode_successes"]) == 20
print("NoiseSweepResult round-trip: PASS")
```

### 1.4.3 7.8.3 Running a Noise Sweep

Recall the sweep definition from Section 7.3: for each $\sigma$ in a list of noise levels, create a `NoisyEvalWrapper` and evaluate the policy for $N$ episodes. The core loop implements this directly, constructing a fresh wrapper per noise level and delegating evaluation to `run_controller_eval`:

```
--8<-- "scripts/labs/robustness.py:run_noise_sweep_loop"
```

??? lab "Full run_noise_sweep function (click to expand)" The complete function includes the signature, docstring, input validation, and the lazy import of `run_controller_eval`: python `--8<-- "scripts/labs/robustness.py:run_noise_sweep"`

This function reuses `run_controller_eval` from the action interface lab (Chapter 6), which means all eight engineering metrics come for free at every noise level. The `extra_wrappers` parameter allows stacking mitigation wrappers (e.g., LowPassFilterWrapper) on top.

!!! lab "Checkpoint" Run a quick sweep with a P-controller to verify SR decreases with noise: "'python from scripts.labs.action_interface import ProportionalController from scripts.labs.robustness import run_noise_sweep

```
controller = ProportionalController("FetchReach-v4", kp=10.0)
results = run_noise_sweep(
    policy=controller, env_id="FetchReach-v4",
    noise_type="obs", noise_levels=[0.0, 0.05, 0.1],
    n_episodes=10, seed=0,
)
for r in results:
    print(f"  sigma={r.noise_std:.3f}: SR={r.success_rate:.0%}")
assert results[0].success_rate >= results[-1].success_rate
print("SR decreasing with noise: PASS")
```

### 1.4.4 7.8.4 Aggregating Across Seeds

The CI formula from Section 7.3:

$$\mathrm{CI}_{95\%} = \overline{x} \pm z \cdot \frac{s}{\sqrt{n}}, \quad z = \begin{cases} 1.96 & n \geq 30 \\ 2.0 & n < 30 \end{cases}$$

--8<-- "scripts/labs/robustness.py:aggregate_across_seeds"

!!! lab ”Checkpoint” Verify that identical seeds give zero variance, different seeds give nonzero variance: “‘python from scripts.labs.robustness import NoiseSweepResult, aggregate_across_seeds

```
r1 = NoiseSweepResult(noise_type="obs", noise_std=0.0, success_rate=1.0,
                      return_mean=-8.0, smoothness_mean=0.01,
                      peak_action_mean=0.9, path_length_mean=0.15,
                      action_energy_mean=25.0, final_distance_mean=0.02,
                      time_to_success_mean=9.0, episode_successes=[True]*10)
r2 = NoiseSweepResult(noise_type="obs", noise_std=0.1, success_rate=0.7,
                      return_mean=-15.0, smoothness_mean=0.05,
                      peak_action_mean=0.95, path_length_mean=0.20,
                      action_energy_mean=30.0, final_distance_mean=0.05,
                      time_to_success_mean=12.0,
                      episode_successes=[True]*7 + [False]*3)

# Two identical seeds -> std = 0
agg = aggregate_across_seeds([[r1, r2], [r1, r2]])
assert agg[0]["success_rate"]["std"] == 0.0
print(f"Identical seeds: std={agg[0]['success_rate']['std']}: PASS")

# Different second seed -> nonzero std
r2_alt = NoiseSweepResult(noise_type="obs", noise_std=0.1, success_rate=0.5,
                          return_mean=-20.0, smoothness_mean=0.08,
                          peak_action_mean=0.98, path_length_mean=0.25,
                          action_energy_mean=35.0, final_distance_mean=0.08,
                          time_to_success_mean=15.0,
                          episode_successes=[True]*5 + [False]*5)
agg2 = aggregate_across_seeds([[r1, r2], [r1, r2_alt]])
assert agg2[1]["success_rate"]["std"] > 0.0
print(f"Different seeds: std={agg2[1]['success_rate']['std']:.4f}: PASS")
```

### 1.4.5 7.8.5 Computing the Degradation Summary

The three summary metrics from Section 7.4 -- critical sigma, degradation slope, and robustness AUC:

$$\sigma^* = \min\{\sigma : \mathrm{SR}(\sigma) < 0.50\}$$

$$\text{slope} = \frac{\sum(\sigma_k - \bar{\sigma})(\mathrm{SR}_k - \overline{\mathrm{SR}})}{\sum(\sigma_k - \bar{\sigma})^2}$$

$$\text{AUC} = \sum_{k=1}^{K} \frac{1}{2}(\text{SR}_{k-1} + \text{SR}_k)(\sigma_k - \sigma_{k-1})$$

The metric computations map directly to these formulas -- critical sigma is a linear scan, slope is a manual linear regression, and AUC is trapezoidal integration:

```
--8<-- "scripts/labs/robustness.py:compute_degradation_summary_metrics"
```

??? lab "Full compute_degradation_summary function (click to expand)" The complete function includes the signature, docstring, and empty-input guard: python --8<-- "scripts/labs/robustness.py:compute_degradation_summary"

!!! lab "Checkpoint" Verify with hand-calculated values from the grounding example in Section 7.4 -- SR = [1.0, 0.8, 0.4] at sigma = [0.0, 0.05, 0.1]: "'python from scripts.labs.robustness import compute_degradation_summary

```
aggregated = [
    {"noise_std": 0.0,  "success_rate": {"mean": 1.0}},
    {"noise_std": 0.05, "success_rate": {"mean": 0.8}},
    {"noise_std": 0.1,  "success_rate": {"mean": 0.4}},
]
summary = compute_degradation_summary(aggregated)

# Critical sigma: SR drops below 50% at sigma=0.1
assert summary["critical_sigma"] == 0.1
print(f"critical_sigma: {summary['critical_sigma']}: PASS")

# Slope should be negative
assert summary["degradation_slope"] < 0
print(f"degradation_slope: {summary['degradation_slope']}: PASS")

# AUC: 0.5*(1.0+0.8)*0.05 + 0.5*(0.8+0.4)*0.05 = 0.075
assert abs(summary["robustness_auc"] - 0.075) < 0.001
print(f"robustness_auc: {summary['robustness_auc']} (expected ~0.075): PASS")

# Edge case: SR never drops below 50%
agg_robust = [
    {"noise_std": 0.0, "success_rate": {"mean": 1.0}},
    {"noise_std": 0.1, "success_rate": {"mean": 0.9}},
]
summary_r = compute_degradation_summary(agg_robust)
assert summary_r["critical_sigma"] is None
print(f"Robust policy -> critical_sigma=None: PASS")
```
```

### 1.4.6 7.8.6 Plugging in Any Controller

A key design property of run_noise_sweep is that it accepts *any* object with a `predict(obs, deterministic) -> (action, _)` interface -- the same interface that SB3 models expose, and also the one implemented by the `ProportionalController` from Chapter 6. This means we can compare a learned SAC policy against a classical P-controller using the exact same evaluation machinery, with no adapter code needed.

!!! lab "Build It (illustrative)" "'python from scripts.labs.action_interface import ProportionalController from scripts.labs.robustness import run_noise_sweep
```

```
# Same function, different policy -- that is the whole trick
controller = ProportionalController("FetchReach-v4", kp=10.0)
results = run_noise_sweep(
    policy=controller,  # <-- not an SB3 model, but same interface
    env_id="FetchReach-v4",
    noise_type="obs",
    noise_levels=[0.0, 0.05, 0.1, 0.2],
    n_episodes=20,
    seed=0,
)
for r in results:
    print(f"  sigma={r.noise_std:.3f}: SR={r.success_rate:.0%}")
```
```

This is what makes the RL-vs-classical comparison in Section 7.13 possible. The chapter script's `cmd_baseline` function does exactly this: it creates a `ProportionalController` and feeds it through the same `run_noise_sweep` calls that the SAC sweeps use. The resulting JSON files have identical structure, so the comparison tables fall out naturally.

!!! lab "Checkpoint" Verify that the PD controller produces valid sweep results with the same structure as an SB3 model would: "'python from scripts.labs.action_interface import ProportionalController from scripts.labs.robustness import run_noise_sweep

```
controller = ProportionalController("FetchReach-v4", kp=10.0)
results = run_noise_sweep(
    policy=controller, env_id="FetchReach-v4",
    noise_type="obs", noise_levels=[0.0, 0.1],
    n_episodes=10, seed=0,
)
assert len(results) == 2
assert results[0].success_rate >= 0.8  # PD solves Reach easily
d = results[0].to_dict()
assert "success_rate" in d and "smoothness_mean" in d
print("PD controller produces valid NoiseSweepResult: PASS")
```
```

### 1.4.7 7.8.7 Verify Everything

Run the built-in verification suite (~60 seconds):

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

Expected output:

```
============================================================
Robustness Lab -- Verification
============================================================
Verifying NoisyEvalWrapper...
  zero noise = identity: OK
  obs noise: observation changed, goals preserved: OK
  action noise with large sigma: env stable: OK
  gym.Wrapper interface: OK
  [PASS] NoisyEvalWrapper OK

Verifying aggregate_across_seeds...
  identical seeds -> mean=1.0, std=0.0: OK
  different seeds -> std=0.1414: OK
  [PASS] aggregate_across_seeds OK

Verifying compute_degradation_summary...
  critical_sigma: 0.1: OK
  degradation_slope: -6.0: OK
  robustness_auc: 0.075 (expected ~0.075): OK
  robust policy -> critical_sigma=None: OK
  [PASS] compute_degradation_summary OK

Verifying run_noise_sweep...
  result count: 3 (expected 3): OK
  sigma=0.000: SR=100%
  sigma=0.050: SR=100%
  sigma=0.100: SR=80%
  [PASS] run_noise_sweep OK

============================================================
[ALL PASS] Robustness lab verified
============================================================
```

---

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

All experiments use checkpoints from Chapter 4 (SAC+HER, FetchReach-v4 and FetchPush-v4), so no new training is needed -- this is an evaluation-only chapter. If you skipped Chapter 4 or are missing checkpoints, the script provides a `train` fallback subcommand that will train SAC+HER automatically:

```bash
bash docker/dev.sh python scripts/ch07_robustness_curves.py train --seeds 0 --include-push
```

The P-controller baseline from Chapter 6 requires no checkpoint at all.

### 1.5.1 7.9 Quick Start: Full Pipeline

```bash
# Full pipeline: obs-sweep + act-sweep + baseline + mitigate + compare (both envs)
bash docker/dev.sh python scripts/ch07_robustness_curves.py all --seeds 0 --include-push
```

This runs the complete robustness analysis -- 6 steps across 2 environments -- producing JSON artifacts in `results/ch07_*.json`. With 20 episodes per condition (smoke test), it takes about 5-10 minutes. For publication-quality results with 100 episodes per condition, add `--n-eval-episodes 100` (roughly 45 minutes).

### 1.5.2 7.10 Experiment 1: SAC Observation Noise Sweep

```bash
bash docker/dev.sh python scripts/ch07_robustness_curves.py obs-sweep --seeds 0 --include-
```

We evaluate the SAC+HER policy at observation noise levels $\sigma_{\text{obs}} \in \{0.0, 0.005, 0.01, 0.02, 0.05, 0.1, 0.2\}$, running 100 episodes per condition.

**FetchReach-v4 -- SAC Observation Noise:**

| $\sigma_{\text{obs}}$ | Success | Return | Smoothness | Dist | TTS |
|---|---|---|---|---|---|
| **0.000** | **100%** | **-1.70** | **0.028** | **0.020** | **2.7** |
| 0.005 | 100% | -1.84 | 0.566 | 0.021 | 2.7 |
| 0.010 | 100% | -3.63 | 1.437 | 0.025 | 2.8 |

| $\sigma_{\text{obs}}$ | Success | Return | Smoothness | Dist | TTS |
|---|---|---|---|---|---|
| 0.020 | 100% | -9.48 | 2.882 | 0.033 | 3.2 |
| 0.050 | 100% | -25.22 | 4.846 | 0.057 | 5.6 |
| 0.100 | 100% | -37.85 | 5.771 | 0.082 | 11.7 |
| 0.200 | **87%** | -45.39 | 6.533 | 0.113 | 17.6 |

**Interpretation:** FetchReach SAC is remarkably robust to observation noise -- 100% success up to $\sigma = 0.1$ (10 cm position error). But extending the range to $\sigma = 0.2$ finally reveals degradation: success drops to 87%. This is the first time Reach-SAC falls below 100% in any of our experiments.

Even where the policy succeeds, movement quality degrades severely. At $\sigma = 0.1$, the gripper takes 11.7 steps to converge (vs 2.7 at zero noise), because noisy observations cause constant overshoot-and-correct cycles. The task is forgiving enough that the policy *eventually* reaches the goal through this jittery process, but at $\sigma = 0.2$ the variance becomes too large for 13% of episodes.

**FetchPush-v4 -- SAC Observation Noise:**

| $\sigma_{\text{obs}}$ | Success | Return | Smoothness | Dist | TTS |
|---|---|---|---|---|---|
| **0.000** | **100%** | **-14.70** | **0.522** | **0.025** | **14.7** |
| 0.005 | 98% | -17.96 | 1.045 | 0.027 | 15.6 |
| 0.010 | 97% | -23.88 | 1.682 | 0.050 | 16.8 |
| 0.020 | 94% | -32.86 | 2.313 | 0.073 | 23.6 |
| 0.050 | **11%** | -46.63 | 3.171 | 0.155 | 18.5 |
| 0.100 | **5%** | -47.50 | 3.499 | 0.200 | 1.0 |
| 0.200 | **5%** | -47.50 | 4.207 | 0.197 | 1.0 |

**Critical sigma:** $\sigma^* = 0.05$ (SR drops from 94% to 11%).

**Interpretation:** Push collapses under observation noise. At $\sigma = 0.05$, only 11% of episodes succeed, and at $\sigma = 0.1$, success is essentially random (5%). The extended range to $\sigma = 0.2$ confirms that Push stays flat at 5% -- once the policy cannot locate the block, adding more noise does not make things worse because the policy is already failing completely. The underlying reason is that Push requires *precise spatial coordination*: the gripper must align with the block, make contact at the right angle, and push in the right direction, so observation noise that corrupts the gripper-to-block distance estimate causes the policy to miss the block entirely or push it in the wrong direction.

### 1.5.3 7.11 Experiment 2: SAC Action Noise Sweep

```bash
bash docker/dev.sh python scripts/ch07_robustness_curves.py act-sweep --seeds 0 --include-
```

We evaluate at action noise levels $\sigma_{\text{act}} \in \{0.0, 0.01, 0.025, 0.05, 0.1, 0.2, 0.35, 0.5\}$.

**FetchReach-v4 -- SAC Action Noise:**

| $\sigma_{\text{act}}$ | Success | Return | Smoothness | Energy | TTS |
|---|---|---|---|---|---|
| **0.000** | **100%** | **-1.70** | **0.028** | **12.4** | **2.7** |
| 0.010 | 100% | -1.69 | 0.029 | 12.5 | 2.7 |
| 0.025 | 100% | -1.69 | 0.031 | 12.5 | 2.7 |
| 0.050 | 100% | -1.69 | 0.037 | 12.8 | 2.7 |

| $\sigma_{\text{act}}$ | Success | Return | Smoothness | Energy | TTS |
|---|---|---|---|---|---|
| 0.100 | 100% | -1.77 | 0.063 | 13.6 | 2.8 |
| 0.200 | 100% | -1.88 | 0.160 | 16.6 | 2.9 |
| 0.350 | 100% | -2.54 | 0.373 | 23.7 | 3.1 |
| 0.500 | 100% | -4.84 | 0.566 | 31.7 | 3.5 |

**Critical sigma: None** (100% success across the entire range).

**Interpretation:** Reach is completely robust to action noise up to $\sigma = 0.5$ -- half the full action range. Even at extreme actuator noise, the policy maintains 100% success, though energy increases from 12.4 to 31.7 (2.6x). Action noise is far less damaging than observation noise because the policy's decision-making (which depends on observations) is unaffected -- only the execution is noisy.

**FetchPush-v4 -- SAC Action Noise:**

| $\sigma_{\text{act}}$ | Success | Return | Smoothness | Energy | TTS |
|---|---|---|---|---|---|
| **0.000** | **100%** | **-14.70** | **0.522** | **34.2** | **14.7** |
| 0.010 | 100% | -14.71 | 0.496 | 33.5 | 14.6 |
| 0.025 | 99% | -16.13 | 0.557 | 36.2 | 15.2 |
| 0.050 | 100% | -15.73 | 0.569 | 36.7 | 15.8 |
| 0.100 | 100% | -15.67 | 0.588 | 38.1 | 15.6 |
| 0.200 | 99% | -17.38 | 0.614 | 40.5 | 16.4 |
| 0.350 | 100% | -20.57 | 0.638 | 44.5 | 18.7 |
| 0.500 | **97%** | -24.95 | 0.766 | 52.0 | 21.1 |

**Critical sigma: None** (never drops below 50%).

**Interpretation:** Push maintains near-perfect success up to $\sigma = 0.35$, with the first crack appearing at $\sigma = 0.5$ (97%). The extended range was worth it -- we can now see the onset of degradation that the old $[0, 0.2]$ range missed entirely. The asymmetry with observation noise remains stark: **Push is brittle to what it sees, but tolerant of what it does.**

### 1.5.4 7.12 Experiment 3: P-Controller Baseline

```bash
bash docker/dev.sh python scripts/ch07_robustness_curves.py baseline --seeds 0 --include-p
```

To understand whether the robustness patterns above are a property of the *learned* policy or the *task*, we run the same noise sweeps with the proportional controller from Chapter 6:

$$a_t = K_p \cdot (g - x_t), \quad K_p = 10$$

where $g$ is the goal position and $x_t$ is the current gripper position. This controller uses the same `predict(obs, deterministic)` interface as SB3 models, so `run_noise_sweep` accepts it directly.

**FetchReach-v4 -- PD Observation Noise:**

| $\sigma_{\text{obs}}$ | Success | Return | Smoothness | Dist | TTS |
|---|---|---|---|---|---|
| **0.000** | **100%** | **-2.08** | **0.006** | **0.002** | **3.1** |
| 0.005 | 100% | -2.15 | 0.026 | 0.005 | 3.1 |
| 0.010 | 100% | -2.19 | 0.086 | 0.007 | 3.2 |

| $\sigma_{\text{obs}}$ | Success | Return | Smoothness | Dist | TTS |
|---|---|---|---|---|---|
| 0.020 | 100% | -2.29 | 0.327 | 0.013 | 3.3 |
| 0.050 | 100% | -8.98 | 1.754 | 0.030 | 4.1 |
| 0.100 | 100% | -30.04 | 3.537 | 0.053 | 6.3 |
| 0.200 | **99%** | -42.88 | 4.765 | 0.086 | 13.8 |

**Interpretation:** The P-controller is nearly perfectly robust to observation noise -- 99% at $\sigma = 0.2$ where SAC dropped to 87%. This is surprising at first glance, but makes sense once we consider the structure of the control law: the P-controller computes $a = K_p(g - x)$, where noise in $x$ adds jitter to the error signal but does not change its *expected direction*. Since the controller has no learned decision boundaries to cross, observation noise cannot cause qualitatively wrong actions -- just noisier convergence. The precision advantage is visible even at zero noise, where the PD controller achieves 0.002 m final distance (vs SAC's 0.020 m).

**FetchReach-v4 -- PD Action Noise:**

| $\sigma_{\text{act}}$ | Success | Return | Smoothness | Energy | TTS |
|---|---|---|---|---|---|
| **0.000** | **100%** | **-2.08** | **0.006** | **3.8** | **3.1** |
| 0.010 | 100% | -2.08 | 0.006 | 3.8 | 3.1 |
| 0.025 | 100% | -2.12 | 0.006 | 3.8 | 3.1 |
| 0.050 | 100% | -2.14 | 0.007 | 3.9 | 3.1 |
| 0.100 | 100% | -2.22 | 0.009 | 4.1 | 3.2 |
| 0.200 | 100% | -2.41 | 0.018 | 4.9 | 3.4 |
| 0.350 | 100% | -3.36 | 0.041 | 7.2 | 3.8 |
| 0.500 | 100% | -8.90 | 0.071 | 10.3 | 4.5 |

**Interpretation:** The PD controller handles action noise even better than SAC -- it uses 3x less energy (10.3 vs 31.7 at $\sigma = 0.5$) because it computes small, precise corrections rather than the larger swings that a neural network policy tends to produce.

**FetchPush-v4 -- PD Observation Noise:**

| $\sigma_{\text{obs}}$ | Success | Return | Smoothness | Dist | TTS |
|---|---|---|---|---|---|
| 0.000 | **5%** | -49.59 | 2.255 | 0.416 | 1.0 |
| 0.005 | 5% | -49.63 | 1.452 | 0.579 | 1.0 |
| 0.010 | 5% | -49.72 | 1.181 | 0.621 | 1.0 |
| 0.020 | 5% | -49.79 | 0.989 | 0.643 | 1.0 |
| 0.050 | 5% | -49.74 | 1.888 | 0.313 | 1.0 |
| 0.100 | 5% | -49.29 | 2.928 | 0.220 | 1.0 |
| 0.200 | 5% | -48.38 | 3.614 | 0.192 | 1.0 |

**FetchPush-v4 -- PD Action Noise:**

| $\sigma_{\text{act}}$ | Success | Return | Smoothness | Energy | TTS |
|---|---|---|---|---|---|
| 0.000 | **5%** | -49.59 | 2.255 | 42.7 | 1.0 |
| 0.010 | 5% | -49.60 | 2.228 | 42.2 | 1.0 |
| 0.025 | 5% | -49.61 | 2.180 | 41.8 | 1.0 |
| 0.050 | 5% | -49.57 | 1.873 | 37.9 | 1.0 |
| 0.100 | 5% | -49.59 | 1.403 | 31.7 | 1.0 |

| $\sigma_{\text{act}}$ | Success | Return | Smoothness | Energy | TTS |
|---|---|---|---|---|---|
| 0.200 | 5% | -49.61 | 0.970 | 27.1 | 1.0 |
| 0.350 | 5% | -49.63 | 0.581 | 24.2 | 1.0 |
| 0.500 | 6% | -49.55 | 0.439 | 24.9 | 5.7 |

**Interpretation:** The PD controller achieves only 5% success on Push at *every* noise level -- including zero noise -- so the degradation curve is flat at 5% because the controller already cannot solve the task. The two-phase approach-then-push strategy is too crude for Push's geometric precision requirements, confirming Chapter 6's finding that Push is a *planning* task beyond the reach of a simple controller. The TTS of 1.0 everywhere is an artifact of this incapability: the only successes are lucky episodes where the block starts near the goal, and these are ”solved” immediately.

### 1.5.5 7.13 Experiment 4: SAC vs P-Controller Side-by-Side

The comparison between SAC and PD reveals the central finding of this chapter -- **learned policies are not just more capable, they are more robust where it matters:**

**Observation Noise -- FetchReach-v4:**

| $\sigma_{\text{obs}}$ | SAC SR | PD SR | SAC Dist | PD Dist |
|---|---|---|---|---|
| 0.000 | 100% | 100% | 0.020 | 0.002 |
| 0.005 | 100% | 100% | 0.021 | 0.005 |
| 0.010 | 100% | 100% | 0.025 | 0.007 |
| 0.020 | 100% | 100% | 0.033 | 0.013 |
| 0.050 | 100% | 100% | 0.057 | 0.030 |
| 0.100 | 100% | 100% | 0.082 | 0.053 |
| 0.200 | **87%** | **99%** | 0.113 | 0.086 |

For Reach under observation noise, the PD controller *beats* SAC at $\sigma = 0.2$ (99% vs 87%), because its linear control law $a = K_p(g - x)$ has no learned features to corrupt. Noise in the position estimate causes jitter, but the expected error direction is always correct, whereas SAC's neural network has sharper input-output mappings that can be more easily disrupted by large perturbations. The PD controller also achieves consistently lower final distance (0.086 vs 0.113 at $\sigma = 0.2$).

**Action Noise -- FetchPush-v4:**

| $\sigma_{\text{act}}$ | SAC SR | PD SR | SAC Dist | PD Dist |
|---|---|---|---|---|
| 0.000 | 100% | 5% | 0.025 | 0.416 |
| 0.100 | 100% | 5% | 0.026 | 0.558 |
| 0.200 | 99% | 5% | 0.025 | 0.580 |
| 0.350 | 100% | 5% | 0.024 | 0.562 |
| 0.500 | **97%** | 6% | 0.039 | 0.530 |

For Push, the comparison is stark: SAC maintains near-perfect success while PD never exceeds 5%. Here, the learned policy's advantage is in *capability*, not noise tolerance -- the PD controller fails even without noise. SAC has learned a complex multi-phase strategy (approach from above, align with block, push toward goal) that no simple controller can replicate. The robustness question only becomes meaningful once the baseline capability exists.

### 1.5.6 7.14 Experiment 5: LPF Mitigation Under Action Noise

```bash
bash docker/dev.sh python scripts/ch07_robustness_curves.py mitigate --seeds 0 --include-p
```

We test the low-pass filter from Chapter 6 as a mitigation strategy under fixed action noise $\sigma_{\text{act}} = 0.5$ -- the highest level in our sweep, where Push finally shows degradation.

**FetchReach-v4 -- Mitigation at $\sigma_{\text{act}} = 0.5$:**

| $\alpha$ | Success | Smoothness | Energy | TTS |
|---|---|---|---|---|
| 0.30 | 100% | 0.435 | 44.8 | 4.1 |
| 0.50 | 100% | 0.478 | 37.3 | 3.6 |
| 0.70 | 100% | 0.509 | 34.0 | 3.5 |
| **1.00** | **100%** | **0.566** | **31.7** | **3.5** |

**Interpretation:** For Reach at $\sigma = 0.5$, LPF filtering does not improve success rate (already 100%) and actually *increases* energy at low alpha. The filter adds lag -- heavier smoothing means the controller takes longer to converge -- without any benefit. This is a useful **negative result**: it tells us that Reach is already robust enough that mitigation is unnecessary, and that blindly applying a filter can make things worse (higher energy, slower convergence) even when it does not hurt success rate.

**FetchPush-v4 -- Mitigation at $\sigma_{\text{act}} = 0.5$:**

| $\alpha$ | Success | Smoothness | Energy | TTS |
|---|---|---|---|---|
| 0.30 | **79%** | 0.530 | 67.2 | 28.4 |
| 0.50 | 92% | 0.594 | 59.9 | 24.6 |
| 0.70 | 94% | 0.612 | 50.1 | 20.8 |
| **1.00** | **97%** | **0.766** | **52.0** | **21.1** |

**Interpretation:** This is where the mitigation experiment becomes interesting. At $\sigma = 0.5$, Push's unfiltered success rate is 97%. Filtering at $\alpha = 0.7$ produces 94%, $\alpha = 0.5$ gives 92%, and heavy filtering at $\alpha = 0.3$ drops to 79% -- an 18-point penalty.

The pattern is clear: **heavy smoothing hurts Push more than the noise it tries to remove.** At $\alpha = 0.3$, the LPF delays the approach-push transition so severely that many episodes time out before completing the push. The filter removes the high-frequency noise but also removes the *intentional* sharp transitions that Push's multi-phase strategy requires. This confirms Chapter 6's finding about the tension between smoothness and responsiveness: for planning tasks, you cannot blindly smooth -- the filter must respect the policy's need for fast mode switches.

### 1.5.7 7.15 Experiment 6: Cross-Environment and Cross-Policy Comparison

Combining all results, we build the complete robustness comparison table:

| Policy | Environment | Noise | Critical $\sigma$ | Slope | AUC |
|---|---|---|---|---|---|
| SAC | FetchReach-v4 | obs | None | -0.59 | 0.194 |
| SAC | FetchReach-v4 | act | None | 0.00 | 0.500 |
| PD | FetchReach-v4 | obs | None | -0.05 | 0.200 |
| PD | FetchReach-v4 | act | None | 0.00 | 0.500 |

| Policy | Environment | Noise | Critical $\sigma$ | Slope | AUC |
|---|---|---|---|---|---|
| SAC | FetchPush-v4 | obs | **0.05** | **-5.39** | 0.044 |
| SAC | FetchPush-v4 | act | None | -0.04 | 0.496 |
| PD | FetchPush-v4 | obs | **0.00** | 0.00 | 0.010 |
| PD | FetchPush-v4 | act | **0.00** | 0.01 | 0.026 |

**Key findings.** The most striking pattern is that observation noise is the critical vulnerability for SAC: both Reach and Push are far more sensitive to observation noise than action noise, with Push-SAC's obs AUC (0.044) running 11x lower than its act AUC (0.496). On the simpler Reach task, the PD controller is actually slightly more robust than SAC under observation noise -- PD achieves AUC = 0.200 (near-perfect over $[0, 0.2]$) while SAC achieves 0.194 (losing 13% at $\sigma = 0.2$) -- because the P-controller's linear structure is inherently resistant to Gaussian input perturbations.

The relationship reverses on Push, where SAC is dramatically more robust than PD. The PD controller starts at 5% success (critical sigma = 0.0) while SAC starts at 100% and only degrades at $\sigma_{\text{obs}} = 0.05$, yielding an AUC ratio of 0.044 / 0.010 = 4.4x -- SAC's learned strategy is far more noise-tolerant *when the task requires planning*. This last comparison also illustrates a deeper point: robustness requires capability first. The PD controller's flat 5% line on Push is not ”robust” in any meaningful sense -- it is uniformly incapable, which means robustness metrics like AUC only become meaningful when baseline performance is high. We can think of the AUC as measuring *how well a policy preserves its capability under perturbation*.

### 1.5.8 7.16 Discussion: What Makes a Policy Brittle?

Stepping back from individual experiments, the data from experiments 1-5 suggests three patterns that connect into a coherent picture of what makes policies fragile.

**Pattern 1: Observation noise is more damaging than action noise.** Both SAC and PD degrade more under observation noise than action noise, which makes sense once we consider what each noise source corrupts: observation noise disrupts the policy's *perception*, causing systematically wrong decisions, while action noise adds *random jitter* around correct decisions. The former introduces bias in the decision-making process; the latter introduces only variance in execution.

**Pattern 2: Planning tasks are more brittle than control tasks.** Push (which requires approach-align-push coordination) degrades faster than Reach (which only requires move-toward-goal), because the *planning* component is what makes Push fragile -- the multi-phase strategy depends on accurate state estimates at each transition between phases, so that even moderate observation noise can cause the policy to mistime or misalign the critical approach-to-push handoff.

**Pattern 3: Linear controllers resist input noise by construction.** The P-controller's robustness on Reach illustrates a general principle: simpler control laws have fewer ”failure modes” under noise, since a linear mapping $a = K_p \cdot e$ cannot be destabilized by Gaussian noise (the expected output direction is always correct). A neural network policy, by contrast, has nonlinear decision boundaries that can be crossed by large perturbations -- which is the classic robustness-capability tradeoff, where the PD controller is more robust on Reach but *fundamentally incapable* on Push.

These three patterns converge on a deeper lesson: robustness is not a single number but depends on the noise type, the task complexity, and the controller architecture. A policy can be simultaneously robust (to action noise) and brittle (to observation noise), or robust on simple tasks and brittle on complex ones, and the degradation curve captures this nuance in a way that a single success rate cannot.

**An open question for Chapter 8:** Does this brittleness pattern transfer to a different simulator? If we test the same policies in a second environment suite, will Push still be the fragile one? Chapter 8 introduces a second simulator adapter to test cross-environment generalization.

The patterns above suggest a natural question: if we know *where* a policy is brittle, can we train one that degrades more gracefully? Sections 7.17 and 7.18 test this with **noise-augmented retraining** -- injecting the same observation noise at training time that we used for diagnosis.

### 1.5.9 7.17 Noise-Augmented Retraining

Diagnosis without treatment is incomplete. The degradation curves above tell us that Push-SAC is brittle to observation noise (critical sigma = 0.05), but they do not tell us whether this brittleness is *inherent* to the task or an artifact of training under ideal conditions. We can test this by retraining SAC+HER with observation noise injected during training itself.

**The idea:** We wrap the training environment with the same `NoisyEvalWrapper` from Section 7.5, but now at *training* time rather than *evaluation* time. The policy sees noisy observations throughout learning, so it must learn a strategy that works despite imprecise state information. The wrapper, noise model, and goal-preservation properties are identical -- only the *phase* changes (training vs evaluation).

**Definition (Noise-Augmented Training).** Training a policy $\pi_\theta$ on the augmented MDP where observations are perturbed: $\tilde{o}_t = o_t + \epsilon$, $\epsilon \sim \mathcal{N}(0, \sigma^2_{\text{train}} I)$. The training noise level $\sigma_{\text{train}}$ is a hyperparameter chosen based on the diagnostic results.

**Choice of $\sigma_{\textbf{train}} = 0.02$:** From the diagnosis, Push-SAC still achieves 94% success at $\sigma_{\text{obs}} = 0.02$, so this is a noise level where the task remains solvable. Training at a level where the task is *mostly* solvable but *slightly* degraded encourages the policy to learn noise-tolerant features without making the optimization problem too hard. If we trained at $\sigma = 0.1$ (where Push collapses to 5%), the policy would have almost no success signal to learn from.

**Definition (Clean-vs-Robust Tradeoff).** Noise-augmented training may improve robustness at the cost of *clean-condition* performance. A policy trained under noise may be slightly less precise under ideal conditions (because it learned conservative strategies) while being significantly more tolerant of perturbations. The tradeoff is quantified by comparing $\text{SR}_{\text{robust}}(\sigma = 0)$ vs $\text{SR}_{\text{clean}}(\sigma = 0)$ (clean-condition cost) against $\text{AUC}_{\text{robust}}$ vs $\text{AUC}_{\text{clean}}$ (robustness gain).

```bash
# Train noise-augmented policy (both envs, 2M steps default)
bash docker/dev.sh python scripts/ch07_robustness_curves.py train-robust \
    --seeds 0 --include-push

# Quick smoke test (Reach only, fewer steps)
bash docker/dev.sh python scripts/ch07_robustness_curves.py train-robust \
    --seeds 0 --robust-total-steps 50000 --no-push
```

The default training budget is 2M steps (`--robust-total-steps 2000000`). Reach converges well before this, but Push needs the full 2M -- our first attempt at 1M reached only 2-6% success rate, while 2M achieved 96% clean eval SR. Training time is roughly ~30 minutes for Reach and ~60-90 minutes for Push on DGX. The noise wrapper adds negligible overhead -- it is just a per-step Gaussian sample added to the observation vector.

### 1.5.10 7.18 Clean vs Robust Comparison

With both checkpoints in hand, we run the same observation noise sweep on both policies and compare side-by-side:

```
# Compare clean vs robust degradation curves
bash docker/dev.sh python scripts/ch07_robustness_curves.py compare-robust --seeds 0 --inc
```

The output is a table showing success rate at each noise level for both policies, plus the three degradation summary metrics.

**FetchReach-v4 (seed 0, 1M robust training steps):**

| $\sigma_{\text{obs}}$ | Clean SR | Robust SR | Delta |
|---|---:|---:|---:|
| 0.0000 | 100% | 100% | +0% |
| 0.0050 | 100% | 100% | +0% |
| 0.0100 | 100% | 100% | +0% |
| 0.0200 | 100% | 100% | +0% |
| 0.0500 | 100% | 100% | +0% |
| 0.1000 | 100% | 100% | +0% |
| 0.2000 | **87%** | **100%** | **+13%** |

**Interpretation:** A free lunch -- the robust policy matches or beats the clean policy at *every* noise level, with zero clean-condition cost. Reach is simple enough that noise-augmented training produces a strictly better policy. The only place it matters is $\sigma = 0.2$, where the clean policy drops to 87% but the robust policy maintains 100%.

**FetchPush-v4 (seed 0, 2M robust training steps):**

| $\sigma_{\text{obs}}$ | Clean SR | Robust SR | Delta |
|---|---:|---:|---:|
| 0.0000 | 100% | **96%** | -4% |
| 0.0050 | 98% | 94% | -4% |
| 0.0100 | 97% | 98% | +1% |
| 0.0200 | 94% | 99% | +5% |
| 0.0500 | **11%** | **74%** | **+63%** |
| 0.1000 | 5% | 12% | +7% |
| 0.2000 | 5% | 5% | +0% |

**Interpretation:** The classic tradeoff -- a 4% clean-condition cost buys dramatically better noise tolerance. The headline result is at $\sigma = 0.05$: the clean policy collapses to 11% while the robust policy maintains 74%, a +63 percentage point improvement. At $\sigma = 0.02$, the robust policy actually *exceeds* the clean policy (99% vs 94%), suggesting that noise-augmented training learned more conservative approach strategies that pay off even at moderate noise levels.

**Degradation summary comparison:**

| Metric | Reach Clean | Reach Robust | Push Clean | Push Robust |
|---|---:|---:|---:|---:|
| Critical $\sigma$ | None | None | **0.05** | **0.10** |
| Degradation slope | -0.59 | 0.00 | -5.39 | -5.33 |
| Robustness AUC | 0.194 | 0.200 | **0.044** | **0.075** |

**What the numbers confirm.** The most important change is that critical sigma doubled for Push ($\sigma^* = 0.05 \rightarrow 0.10$), meaning the robust policy tolerates twice the noise before hitting the 50% threshold. Interestingly, the degradation slope barely changed (-5.39 vs -5.33), which tells us the cliff is just as steep -- noise-augmented training shifted it *rightward* rather than making it gentler, so that once the robust policy starts failing, it fails just as fast. Integrating across all noise levels, AUC improved 71% for Push (0.044 $\rightarrow$ 0.075), a substantial gain in overall noise tolerance. The clean-condition cost for this improvement is 4% on Push (100% $\rightarrow$ 96%) -- a modest price for the robustness gain -- while for Reach the clean-condition cost is zero.

**The diagnose-treat-verify arc:** This completes Chapter 7's narrative. We started by *diagnosing* brittleness (Sections 7.10-7.15), then *treated* it with noise-augmented training (Section 7.17), and now *verified* the improvement quantitatively (this section). The degradation curve serves as both the diagnostic tool and the verification metric -- the same measurement, applied before and after treatment.

---

## 1.6 Summary

### 1.6.1 Concepts Introduced in This Chapter

| Concept | Definition |
|---|---|
| Observation noise | Gaussian perturbation of sensor readings: $\tilde{o}_t = o_t + \mathcal{N}(0, \sigma^2 I)$ |
| Action noise | Gaussian perturbation of commanded actions: $\tilde{a}_t = \text{clip}(a_t + \mathcal{N}(0, \sigma^2 I))$ |
| Degradation curve | Success rate as a function of noise level: $\sigma \mapsto \text{SR}(\sigma)$ |
| Critical sigma ($\sigma^*$) | First noise level where SR drops below 50% |
| Degradation slope | Linear regression slope of SR vs $\sigma$ (more negative = more brittle) |
| Robustness AUC | Area under the SR-vs-$\sigma$ curve (higher = more robust) |
| Noise injection | Eval-time wrapper that adds controlled noise without retraining |
| Brittleness fingerprint | The full degradation curve characterizing a policy's noise tolerance |
| Noise-augmented training | Training with observation noise injected to improve robustness |
| Clean-vs-robust tradeoff | Possible clean-condition cost vs noise-tolerance gain from noise-augmented tra |

### 1.6.2 Files Generated

| File | Purpose |
|---|---|
| `scripts/labs/robustness.py` | Pedagogical toolkit: wrapper, sweep, aggregation, summary (Bu |
| `scripts/ch07_robustness_curves.py` | Chapter script: obs-sweep, act-sweep, baseline, mitigate, train- |
| `tutorials/ch07_robustness_curves.md` | This tutorial |

### 1.6.3 Artifacts

| Artifact | Location |
|---|---|
| SAC obs sweep results | `results/ch07_obs_sweep_{env}_seed{N}.json` |
| SAC act sweep results | `results/ch07_act_sweep_{env}_seed{N}.json` |
| PD baseline obs sweep | `results/ch07_baseline_obs_sweep_{env}_seed{N}.json` |
| PD baseline act sweep | `results/ch07_baseline_act_sweep_{env}_seed{N}.json` |
| Mitigation results | `results/ch07_mitigation_{env}_seed{N}.json` |
| Comparison report | `results/ch07_comparison.json` |
| Robust checkpoint | `checkpoints/sac_her_{env}_seed{N}_robust.zip` |
| Robust metadata | `checkpoints/sac_her_{env}_seed{N}_robust.meta.json` |
| Clean vs robust report | `results/ch07_clean_vs_robust_{env}.json` |

### 1.6.4 What Comes Next

We now have a rich quantitative picture of policy brittleness -- and a demonstrated treatment. Push-SAC is fragile to observation noise but robust to action noise; the PD controller is more robust on Reach but incapable on Push; mitigation via low-pass filtering has sharp limits on planning tasks; and noise-augmented retraining shifts the critical sigma rightward, trading a small clean-condition cost for measurably improved noise tolerance.

The clean-vs-robust comparison also provides a useful baseline for Chapter 8: when we test the same policies in a different simulator, we can compare whether the robustness *improvements* from noise-augmented training transfer across environments. Chapter 8 introduces a **second environment suite adapter** to test whether these robustness patterns hold across different physics engines -- the first step toward understanding sim-to-sim transfer.