Proximal Policy Optimization for Safe Drug Concentration Control-From Theory to a Working Simulation

20 min read · 21 hours ago

Sheng Jiang

github link

https://medium.com/@shengj1ang/proximal-policy-optimization-for-safe-drug-concentration-control-from-theory-to-a-working-5eb8669a48e0

1. Introduction

Reinforcement learning (RL) has become a powerful framework for sequential decision-making, enabling agents to learn control behaviour through trial-and-error interaction rather than labelled datasets [1–4]. Its flexibility has driven applications across robotics, autonomy, healthcare decision support, energy systems, and large-scale control [5]. A major advantage of RL is its ability to learn policies for dynamic and uncertain systems with long-term objectives and continuous action spaces, where classical optimisation or manual tuning is often impractical [6].

In this article, I introduce Proximal Policy Optimisation (PPO) through a drug-concentration control task, where an agent must regulate blood concentration near a therapeutic target while limiting excessive dosing. The key goal is methodological: the PPO algorithm is kept fixed, and its behaviour is examined under a family of

progressively more challenging pharmacokinetic (PK) simulators — from simple linear dynamics to continuous-time ODE models and multi-compartment settings.

The environment is designed as a configurable testbed that enables systematic stress-testing of PPO under increasing realism. Across variants, the agent selects bounded infusion actions in [0,2], while dynamics may include multi-compartment coupling with partial observability (central concentration observed, peripheral state hidden), nonlinear elimination effects, and stochastic process and observation noise. Pharmacokinetic parameters can also be randomly varied to represent patient-to-patient heterogeneity. This staged design allows evaluation of PPO in settings where stability, robustness, and penalty-based safety considerations become as important as reward maximisation.

Beyond explaining the core ideas behind PPO, this article serves as a hands-on engineering walkthrough. I demonstrate an actor–critic implementation stabilised with practical choices including generalised advantage estimation (GAE) for lower-variance learning signals, tanh-squashed Gaussian actions to enforce dosing bounds, and gradient clipping and entropy regularisation for robust optimisation.

Finally, to connect learning-based control with classical intuition, I include a PID baseline comparison, highlighting when hand-tuned controllers can be sufficient and where a learned policy becomes more adaptable as dynamics, noise, and patient characteristics vary.

2. Theoretical Explanation

2.1. Problem Formulation as an MDP

To apply reinforcement learning to drug dosing, we model the task as a Markov Decision Process (MDP) defined by (S,A,P,r,γ). At each discrete time step t, the agent observes the patient state $s_t \in S$, selects a dosing action $a_t \in A$, and the environment transitions to a new state s_{t+1} according to the dynamics $P(s_{t+1}|s_t,a_t)$. The agent receives a scalar reward $r_t = r(s_t,a_t)$ that encourages therapeutic control while penalising unsafe or inefficient dosing. The objective is to learn a policy π_θ

(a|s) that maximises the expected discounted return:

$$J(\pi_\theta) = \mathbb{E}\left[\sum_{t=0}^{T-1} \gamma^t r_t\right],$$

where $\gamma \in (0,1]$ is the discount factor and T is the episode length
In the first (prototype) version, the environment was a simplified 1D concentration model where the state effectively reduced to a single concentration variable. In the improved version, I upgrade the internal pharmacokinetics to a 2D (two-compartment) model, where the underlying state can be written as:

$$s_t = (c_{1,t}, c_{2,t}),$$

with c_1 representing the central compartment concentration and c_2 the peripheral compartment concentration.Importantly, in many realistic settings we do not directly measure every compartment. Therefore, I keep the observation minimal (e.g., only c_(1,t)) while the environment dynamics remain 2D. This makes the task more realistic: the agent must control dosing based on partial information while hidden dynamics still influence future concentration.
To reflect inter-patient variability, model parameters are randomly sampled at the beginning of each episode to represent different clearance profiles (e.g., slow vs. fast metabolism). PPO therefore learns under heterogeneous dynamics rather than a single nominal patient.

Different environment instantiations are used in this work, ranging from a minimal single-state prototype to a two-compartment pharmacokinetic model. All are formulated under the same MDP (or POMDP) framework to isolate algorithmic effects from modelling complexity.

2.2. Action Space and Safety Constraints

The action a_t represents a continuous infusion (or dosing) rate. In practice, dosing must satisfy physical and safety limits, so the environment applies bounds:

$$a_t \in [a_{min}, a_{max}].$$

In PPO implementations, it is important to distinguish the raw action sampled from the policy distribution (used to compute log-probabilities) from the environment action after tanh squashing (used to update the patient state). This separation prevents inconsistencies in the policy-gradient update when actions are bounded.

This separation is particularly important in policy-gradient methods such as PPO, as it allows stable likelihood-ratio updates while enforcing hard physical constraints on dosing actions.

2.3. Reward Design as a Control Objective

Drug dosing is fundamentally a tracking control problem: keep the central concentration c_(1,t) near a therapeutic target c^* (e.g., normalised to 1), while avoiding unnecessary drug usage. Therefore, I use a reward that combines a tracking term and an action-effort penalty:

$$r_t = -\alpha\left(c_{1,t} - c^*\right)^2 - \beta a_t^2.$$

set alpha=6 and beta=0.01

The quadratic tracking term provides a smooth learning signal, encouraging the agent to reduce concentration error. The action penalty discourages overly aggressive dosing and helps produce clinically plausible policies. In my earlier prototype, the reward scaling and action penalty could dominate early learning and accidentally encourage a degenerate “zero-action” solution. Increasing the weight on concentration tracking (and choosing a small but non-zero β) improves the

learning signal and prevents collapse to “do nothing”.
In my implementation, the policy outputs bounded actions via tanh squashing, and the environment additionally clips actions as a safety guard.

2.4 Actor-Critic: Value Function and Advantage Estimation

In this setting, GAE is particularly useful because reward signals are noisy and delayed due to pharmacokinetic dynamics, making low-variance advantage estimates critical for stable learning.

PPO is an actor-critic method. The actor (policy) $\pi_\theta(a|s)$ outputs a distribution over continuous actions (commonly Gaussian), while the critic approximates the state value $V_\phi(s)$, which estimates the expected return from state s. The advantage function measures how much better an action is compared to the baseline value:

$$A_t = Q(s_t, a_t) - V(s_t).$$

In practice, PPO often uses Generalised Advantage Estimation (GAE) to reduce variance while maintaining low bias. With TD residuals

$$\delta_t = r_t + \gamma V_\phi(s_{t+1}) - V_\phi(s_t),$$

GAE computes:

$$\widehat{A}_t = \sum_{l=0}^{T-t-1} (\gamma\lambda)^l \delta_{t+l},$$

where $\lambda \in [0,1]$ controls the bias–variance trade-off.

2.5 PPO Clipped Objective: Why Updates Stay Stable

Vanilla policy gradient methods can become unstable if a single update changes the policy too much. PPO addresses this by optimising a clipped surrogate objective. Let

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)}.$$

PPO maximises:

$$L^{CLIP}(\theta) = \mathbb{E}\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right],$$

where ϵ is a small clipping parameter (e.g., 0.1–0.3). Intuitively, if the new policy probability ratio r_t tries to move outside [1-ϵ,1+ϵ], the objective stops encouraging further movement. This prevents overly large policy updates and improves training robustness. The full PPO loss typically combines policy loss, value-function loss, and an entropy bonus:

$$L = -L^{CLIP} + c_v L^V - c_e H(\pi_\theta),$$

where

is a mean-squared error for critic training and encourages exploration.

Although the underlying pharmacokinetics evolve in continuous time according to an ODE system, the control problem is discretised at fixed intervals and treated as a Markov decision process for learning. In the two-compartment setting, the agent

only observes the central compartment concentration, while peripheral dynamics remain hidden. As a result, the task is more accurately described as a partially observable MDP (POMDP), with PPO learning a reactive policy based on noisy observations.

3. Code Demonstration

This section walks through the full PPO pipeline used in this work, from defining the actor–critic networks and bounded action sampling to computing GAE advantages and performing the clipped PPO update. The drug-concentration task is used only as a lightweight testbed to demonstrate the algorithm end-to-end across different environment variants; details of the environment models are kept brief here. For readability, only key code fragments are shown in the text — the complete, runnable implementation (including training loop and environment configurations) is available in the accompanying GitHub repository.

3.1 Environment Overview

The experiments are conducted in a lightweight pharmacokinetic (PK) control environment designed to serve as a testbed for evaluating PPO rather than as a detailed physiological model. The environment supports both a minimal one-compartment prototype and a simplified two-compartment formulation, allowing environment complexity to be increased while keeping the learning algorithm unchanged.

In the two-compartment setting, the true system state consists of a central and a peripheral drug concentration, while the agent only observes the central compartment. This introduces partial observability, reflecting the fact that not all internal pharmacokinetic states are directly measurable in practice. The agent selects a continuous infusion rate bounded to [0,2], and system dynamics are advanced using a discrete-time Euler update. Optional extensions include nonlinear elimination effects, stochastic process and observation noise, and per-episode parameter variation to represent inter-patient differences in drug clearance.

For clarity and reproducibility, a discrete-time Euler-integrated version of the PK

model is presented alongside the PPO implementation. The full experimental results are obtained using a continuous-time ODE-based formulation, but both environments share the same key characteristics: bounded continuous actions, noisy one-dimensional observations, hidden internal dynamics, and a quadratic tracking-based reward. This design ensures that observed differences in performance can be attributed primarily to the behaviour of PPO rather than to changes in environment structure.

For brevity, the listing shows a minimal Euler-discretised two-compartment PK testbed with partial observability; the full ODE-based environment and additional options are provided in the accompanying GitHub repository.

```
class PKEnv(gym.Env):
    """
    Minimal PK control testbed (Euler-discretised).
    - True state: x = [c1, c2] (central, peripheral)
    - Observation: o = [c1] only (partial observability)
    - Action: u in [0, 2] (bounded infusion rate)
    - Options: process/observation noise, per-episode patient variability
    """
    metadata = {"render_modes": []}

    def __init__(self, dt=0.1, use_noise=True, use_patient_variability=True):
        super().__init__()
        self.dt = dt
        self.use_noise = use_noise
        self.use_patient_variability = use_patient_variability

        # Target and default PK parameters (overwritten per episode if variabil
        self.target = 1.0
        self.k10, self.k12, self.k21 = 0.3, 0.2, 0.1

        # Noise levels
        self.process_noise_std = 0.01
        self.obs_noise_std = 0.05

        # RL interface
        self.action_space = gym.spaces.Box(
            low=np.array([0.0], dtype=np.float32),
            high=np.array([2.0], dtype=np.float32),
        )
        self.observation_space = gym.spaces.Box(
```

```
            low=np.array([0.0], dtype=np.float32),
            high=np.array([5.0], dtype=np.float32),
        )

        self.state = None  # [c1, c2]

    def reset(self, seed=None, options=None):
        super().reset(seed=seed)

        # Two patient types: slower vs faster clearance/distribution
        if self.use_patient_variability:
            slow = (np.random.rand() < 0.5)
            self.k10, self.k12, self.k21 = (0.15, 0.10, 0.05) if slow else (0.8

        # Initial concentrations (arbitrary but consistent)
        self.state = np.array([0.5, 0.3], dtype=np.float32)
        return self._observe(), {}

    def _observe(self):
        c1 = float(self.state[0])
        noise = np.random.normal(0.0, self.obs_noise_std) if self.use_noise els
        obs = np.array([c1 + noise], dtype=np.float32)
        return np.clip(obs, self.observation_space.low, self.observation_space.

    def step(self, action):
        # Bound action (infusion rate)
        u = float(np.clip(action[0], 0.0, 2.0))

        c1, c2 = float(self.state[0]), float(self.state[1])

        # Euler update of a linear 2-compartment PK model
        dc1 = (-(self.k10 + self.k12) * c1 + self.k21 * c2 + u) * self.dt
        dc2 = ( self.k12 * c1 - self.k21 * c2) * self.dt
        new_state = np.array([c1 + dc1, c2 + dc2], dtype=np.float32)

        # Process noise (applied to hidden true state)
        if self.use_noise:
            new_state += np.random.normal(0.0, self.process_noise_std, size=new

        self.state = new_state

        # Reward: quadratic tracking + control effort
        c1 = float(self.state[0])
        error = c1 - self.target
        reward = -(6.0 * error * error) - 0.01 * (u * u)

        obs = self._observe()
        terminated, truncated = False, False
```

```
        info = {"true_state": self.state.copy(), "central_concentration": c1}
        return obs, reward, terminated, truncated, info
```

3.2 Policy & Value Networks

In my implementation, the PPO agent uses two neural networks: a policy network that outputs Gaussian parameters (μ, σ) and a value network V(s). I also set a global learnable log-std (clamped for stability), which keeps early exploration active and avoids collapsing to near-deterministic actions too early.

```
class Policy(nn.Module):
    """
    Policy network:
    - Input: noisy concentration (1D)
    - Output: Gaussian parameters (mu, std) for an unconstrained action z
    - Environment action is: a = 1 + tanh(z)  in [0, 2]
    """
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(1, 64),
            nn.ReLU(),
            nn.Linear(64, 64),
            nn.ReLU()
        )
        self.mu = nn.Linear(64, 1)
        # Global log_std parameter, clamped during forward for stability
        self.log_std = nn.Parameter(torch.tensor([-0.7]))  # around std ≈ 0.5
    def forward(self, x):
        """
        Forward pass:
        - x: [batch, 1]
        Returns:
        - mu: [batch, 1]
        - std: [batch, 1]
        """
        x = self.net(x)
        mu = self.mu(x)
        # Clamp log_std to avoid extremely small/large std
        log_std = torch.clamp(self.log_std, -2.0, 2.0)
        std = log_std.exp()
        return mu, std
class Value(nn.Module):
    def __init__(self):
```

```
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(1, 64),
            nn.ReLU(),
            nn.Linear(64, 64),
            nn.ReLU(),
            nn.Linear(64, 1)
        )
    def forward(self, x):
        return self.net(x)
```

3.3 Sampling

To enforce bounded dosing actions while preserving a simple and stable PPO implementation, actions are sampled in two stages. First, the policy samples an unconstrained Gaussian variable $z \sim \mathcal{N}(\mu, \sigma)$. This raw action is then squashed using a hyperbolic tangent and linearly mapped to the valid action range via a = 1 + tanh(z) in [0,2].

In the implementation, the log-probability required for the PPO likelihood ratio is

computed with respect to the unsquashed variable z, while the squashed action is used only to interact with the environment. This separation is a common practical simplification in continuous-control PPO implementations and keeps the likelihood-ratio computation straightforward and numerically stable when enforcing hard action bounds.

```
def select_action(self, state):
    """
    Select an action given current (noisy) observation.
    Args:
        state: np.array([concentration])
    Returns:
        env_action: np.array([a]) in [0, 2], used in environment
        raw_action: tensor([z]), unconstrained Gaussian sample (before tanh)
        logp:       tensor scalar, log_prob of raw_action (over z)
    """
    s = torch.tensor(state, dtype=torch.float32).unsqueeze(0)  # [1,1]
    mu, std = self.policy(s)
    dist = torch.distributions.Normal(mu, std)

    # z is unconstrained action
    z = dist.rsample()  # [1,1], reparameterization trick
    # Squash to [-1, 1]
    a_tanh = torch.tanh(z)
    # Map to [0, 2]
    env_a = 1.0 + a_tanh  # center at 1.0, radius 1.0
    # Log probability of z under the Gaussian
    logp = dist.log_prob(z).sum(dim=-1)  # [1]
    return env_a.detach().numpy()[0], z.detach()[0], logp.detach()[0]
```

3.4 Advantage Estimation with GAE(λ)

Instead of using a simple return-minus-baseline estimate, advantages are computed using Generalised Advantage Estimation (GAE(λ)). GAE provides a low-variance learning signal by exponentially weighting multi-step temporal-difference residuals, which is particularly important in pharmacokinetic control tasks where rewards are delayed and noisy.

Episodes are truncated at a fixed horizon rather than terminating at a true terminal state. Therefore, the value function is bootstrapped at the final time step using

V(s_T) when computing advantages. After computing the advantage estimates, they are normalised within each batch to improve numerical conditioning and stabilise PPO updates.

```
def compute_returns_and_advantages(self, rewards, states, last_state):
    """
    Use GAE(λ) to compute returns and advantages.
    """
    with torch.no_grad():
        # values for all states in the episode: [T, 1] -> [T]
        values = self.value(states).squeeze(-1)

        # bootstrap value for the final state
        last_state_t = torch.tensor(last_state, dtype=torch.float32).unsqueeze(
        last_value = self.value(last_state_t).squeeze()  # scalar (0D tensor)

    # extended values: [T+1]
    last_value = last_value.view(1)  # [1]
    values_ext = torch.cat([values, last_value], dim=0)  # [T+1]

    T = len(rewards)
    advantages = torch.zeros(T)
    gae = 0.0

    # GAE(λ) backward recursion
    for t in reversed(range(T)):
        delta = rewards[t] + self.gamma * values_ext[t + 1] - values_ext[t]
        gae = delta + self.gamma * self.lam * gae
        advantages[t] = gae

    returns = advantages + values
    advantages = (advantages - advantages.mean()) / (advantages.std() + 1e-8)
return returns, advantages
```

3.5 PPO Update

Policy optimisation follows the standard PPO clipped objective. Using trajectories collected by the current policy, the update computes the likelihood ratio between the new and old policies, applies clipping to limit the magnitude of policy changes, and combines the resulting policy loss with a value-function loss and an entropy regularisation term.

For each batch of on-policy data, multiple optimisation epochs are performed over the same trajectory to improve sample efficiency. During each update, gradients are clipped to a fixed norm to prevent unstable parameter updates. This combination of clipping in the objective, entropy regularisation, and gradient norm clipping provides robust and stable learning behaviour across different environment configurations.

```
def update(self, states, raw_actions, logp_old, rewards, last_state):
    """
    Perform PPO update using data from one episode.
    """
    states = states.detach()
    raw_actions = raw_actions.detach()
    logp_old = logp_old.detach()
    returns, advantages = self.compute_returns_and_advantages(rewards, states,
    total_loss = 0.0
    for _ in range(self.ppo_epochs):
        mu, std = self.policy(states)
        dist = torch.distributions.Normal(mu, std)
        new_logp = dist.log_prob(raw_actions).sum(dim=-1)  # [T]
        entropy = dist.entropy().sum(dim=-1).mean()

        ratio = torch.exp(new_logp - logp_old)  # [T]
        surr1 = ratio * advantages
        surr2 = torch.clamp(ratio, 1 - self.eps_clip, 1 + self.eps_clip) * adva
        policy_loss = -torch.min(surr1, surr2).mean()
        values = self.value(states).squeeze(-1)
        value_loss = F.mse_loss(values, returns)
        loss = policy_loss + self.value_coef * value_loss - self.entropy_coef *
        self.optimizer.zero_grad()
        loss.backward()
        # Gradient clipping for stability
        torch.nn.utils.clip_grad_norm_(
            list(self.policy.parameters()) + list(self.value.parameters()),
            self.max_grad_norm
        )
        self.optimizer.step()
        total_loss += loss.item()
    return total_loss / self.ppo_epochs
```

4. Results and Visualisation

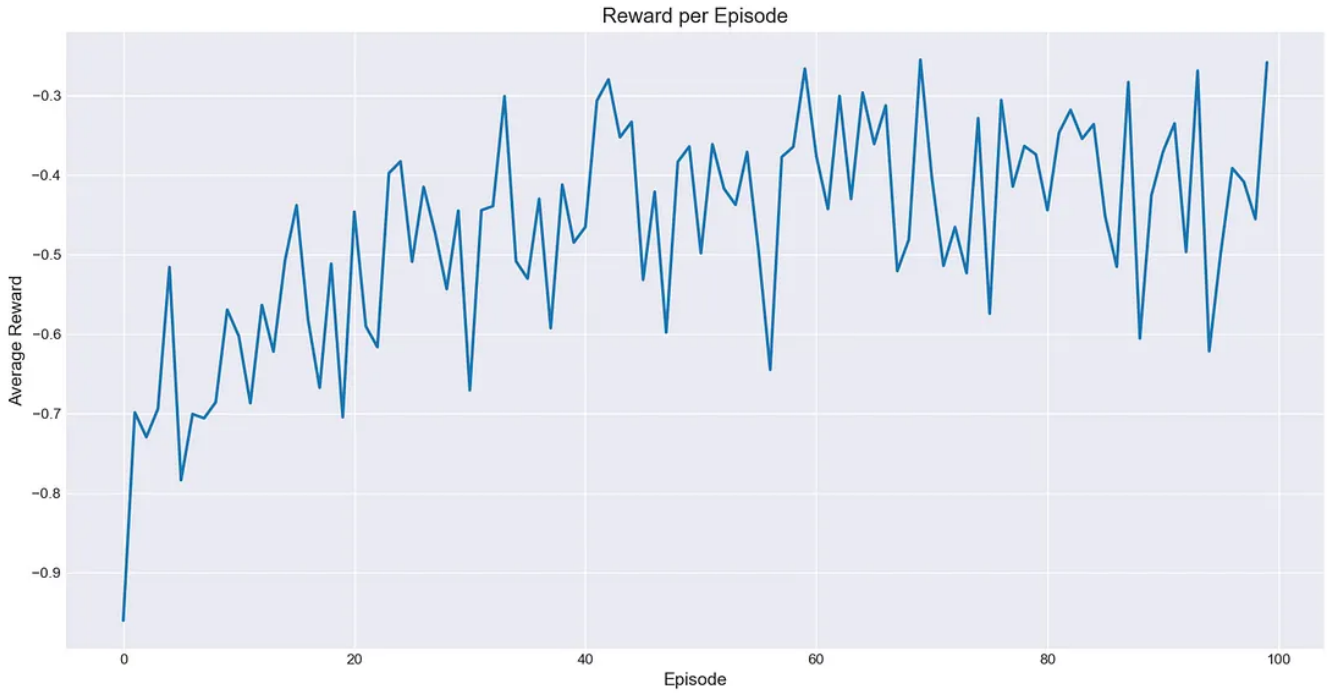


Figure 1. Average Reward per Episode

Figure 1 shows the average reward per episode during training. Initially, the reward is around -1 due to the agent's random policy and large deviation from the target concentration. As training progresses, the PPO agent gradually learns to adjust injection amounts to maintain blood concentration near the target, and the average reward stabilizes around -0.4. This upward trend indicates effective policy learning.

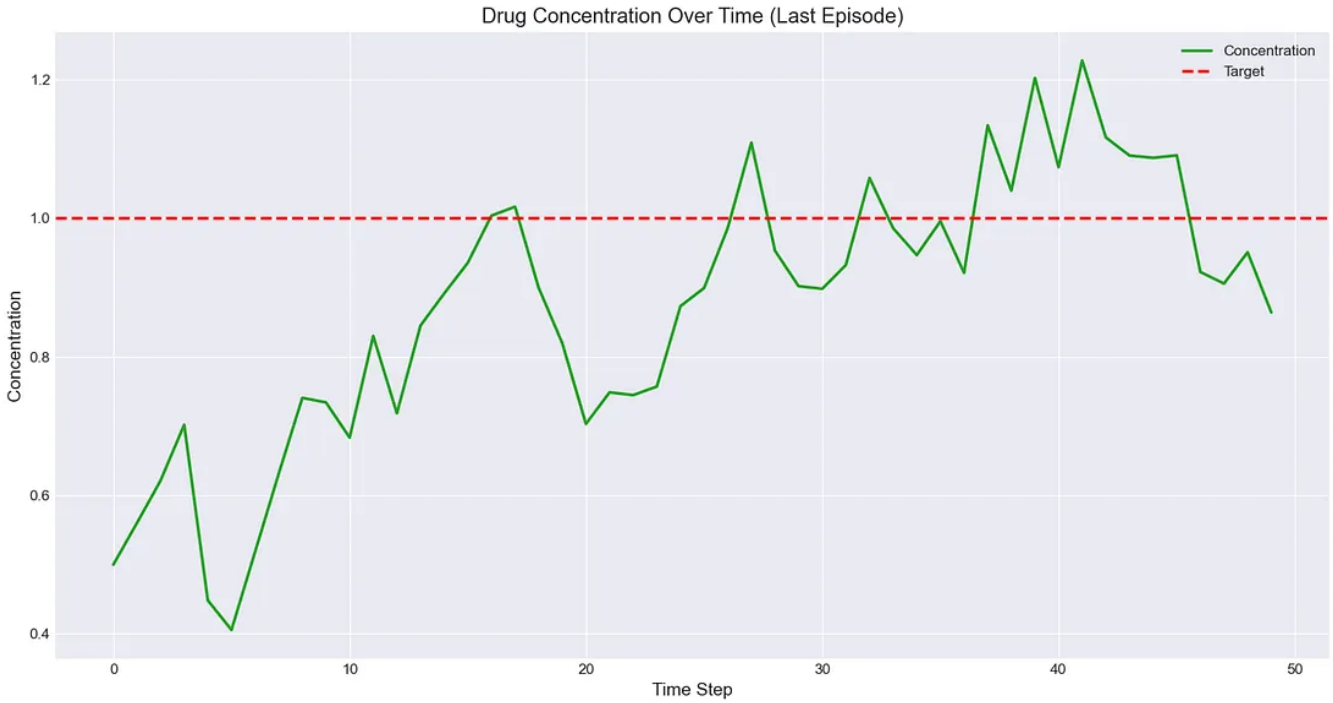


Figure 2. Blood Concentration Over Time (Last Episode)

Figure 2 illustrates the blood concentration over time in the last training episode. Early time steps may show fluctuations, but later the concentration stabilizes around the target value (red dashed line). The smoothness and closeness to the target demonstrate that the agent has learned a reasonable injection policy, achieving stable control of the desired blood concentration.

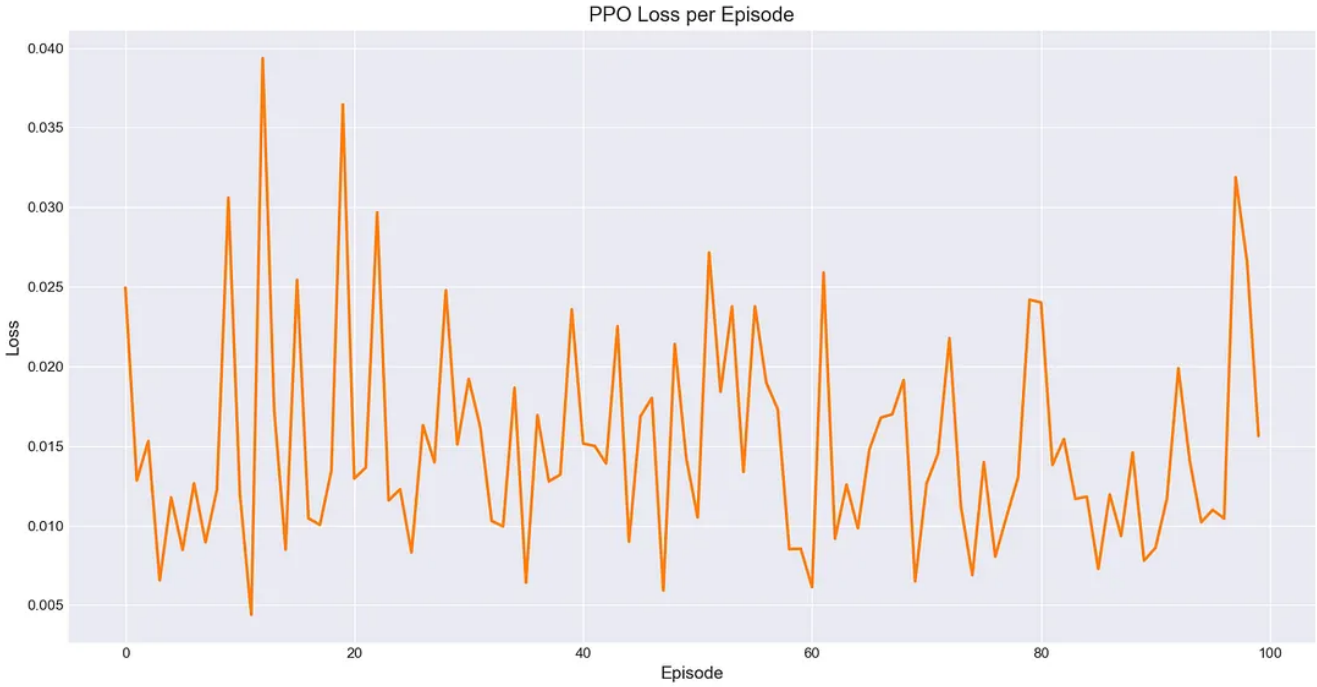


Figure 3. PPO Loss per Episode

Figure 3 depicts the PPO loss per episode. Unlike the reward curve, the loss does not decrease monotonically. Instead, it exhibits high fluctuations: some episodes show very high loss values, while others remain closer to normal levels. This behavior reflects the inherent variability in policy updates during PPO training, as the clipping mechanism and stochastic sampling can cause occasional spikes. These fluctuations are likely caused by the stochastic nature of PPO updates, including

1. Mini-batch sampling variability: Each policy update uses a subset of experiences, which can produce varying gradient magnitudes;
2. Clipping in the PPO objective: The clipping mechanism prevents large policy updates, but can occasionally lead to abrupt changes in loss values depending on advantage estimates;
3. Sparse or noisy rewards in some episodes: Small deviations in blood concentration can result in large negative rewards, temporarily increasing loss.

Building on the 1D prototype results above, I increased the complexity of the study by (i) correcting key PPO design/implementation issues (Version1) and (ii) extending

the control task towards a higher-dimensional (2D) pharmacokinetic setting; the following results summarise these improvements and their impact on learning stability.

4.1 Debugging the Prototype: Summary of Improvements

The original prototype frequently converged to a degenerate "zero-action" policy (inject nothing). After analysing reward shaping, action sampling, and the PPO update pipeline, I identified three core problems and applied targeted fixes.

Problem 1 — Incentive structure: the original reward made "doing nothing" an easy local optimum because the action penalty dominated early learning.

Problem 2 — Inconsistent action sampling/log-probabilities: clipping actions to [0,2] caused many samples to collapse to 0, and computing log-probabilities after clipping can break PPO's gradient consistency.

Problem 3 — Unstable advantage/value learning: high-variance returns and an unstable value baseline reduced the quality of the policy gradient signal.

Key Fix 1 (critical) — Stronger reward shaping: I switched to a quadratic tracking reward with a smaller action penalty, $r = -6(c_t - 1)^2 - 0.01\, a_t^2$, which provides a smoother and stronger learning signal toward the target concentration.

Key Fix 2 — Raw action vs environment action separation: log-probabilities are computed on the raw Gaussian action before any squashing/clipping, while only the bounded action is applied to the environment.

Key Fix 3 — Exploration and stability: I increased early exploration via an initial policy standard deviation around 0.5 ($\log\sigma \approx -0.7$) and used standard PPO components (clipped objective, value loss, entropy bonus, multiple epochs per rollout).

4.2 Results Comparison: Prototype vs Version1

Figures 4–6 show that Version1 achieves a much more stable reward progression and improved concentration tracking, while Figures 7–8 illustrate the prototype's degenerate behaviour and unstable loss scale. Overall, the fixes lead to a more meaningful learning signal and more reliable PPO optimisation.

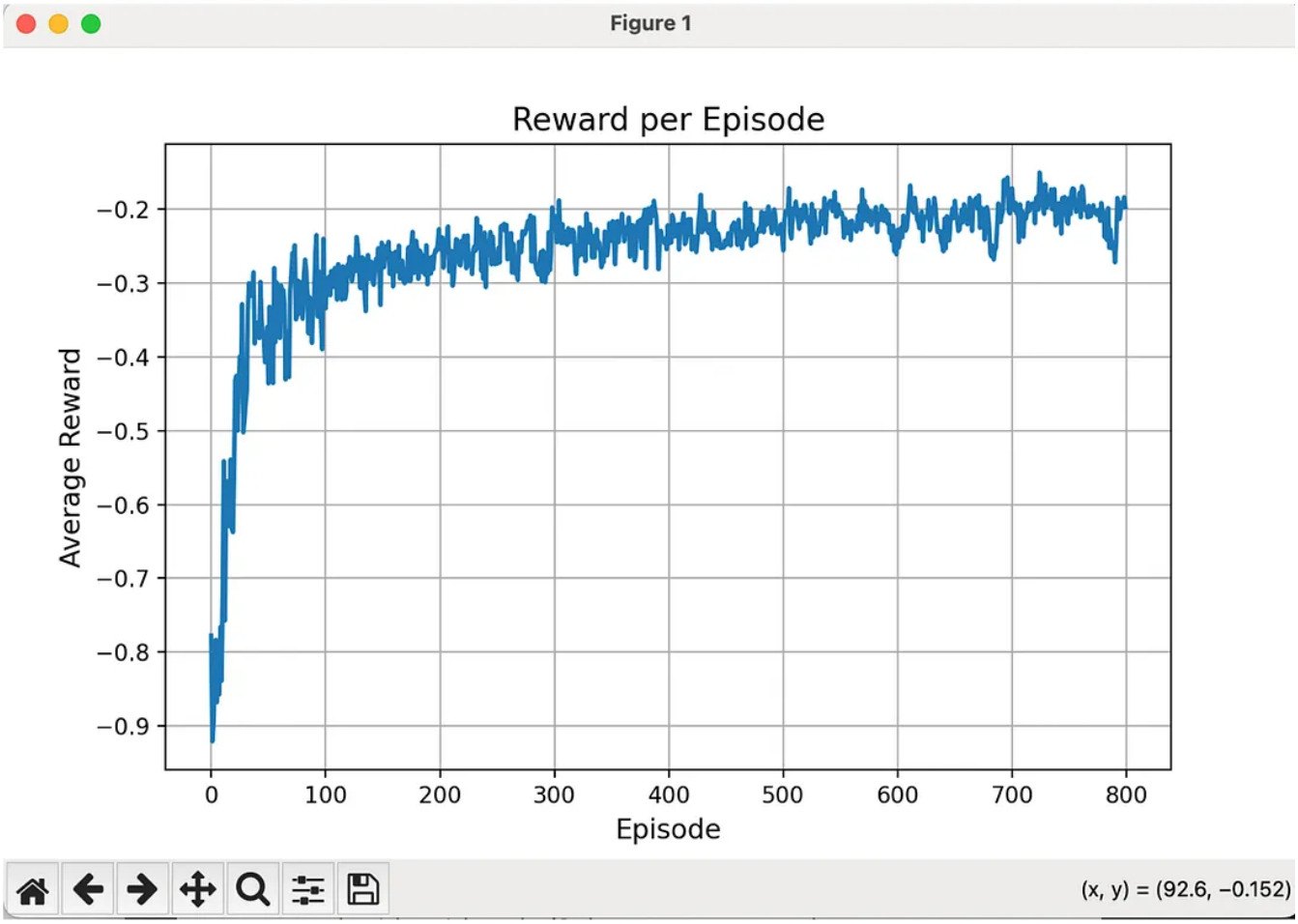


Figure 4. Reward per Episode after fixes (Version1)

Figure 4 shows that, after applying the fixes, the average reward per episode no longer collapses to a highly negative plateau. Instead, the curve exhibits a clear upward trend and then stabilises around a much higher level compared with the original prototype. Occasional fluctuations are still present, which is expected in PPO due to stochastic updates, but overall the training signal becomes smoother and more informative. This indicates that the revised reward shaping and action handling successfully steer the agent away from the degenerate “do nothing” solution and allow it to discover policies that maintain the drug concentration much closer to the therapeutic target.

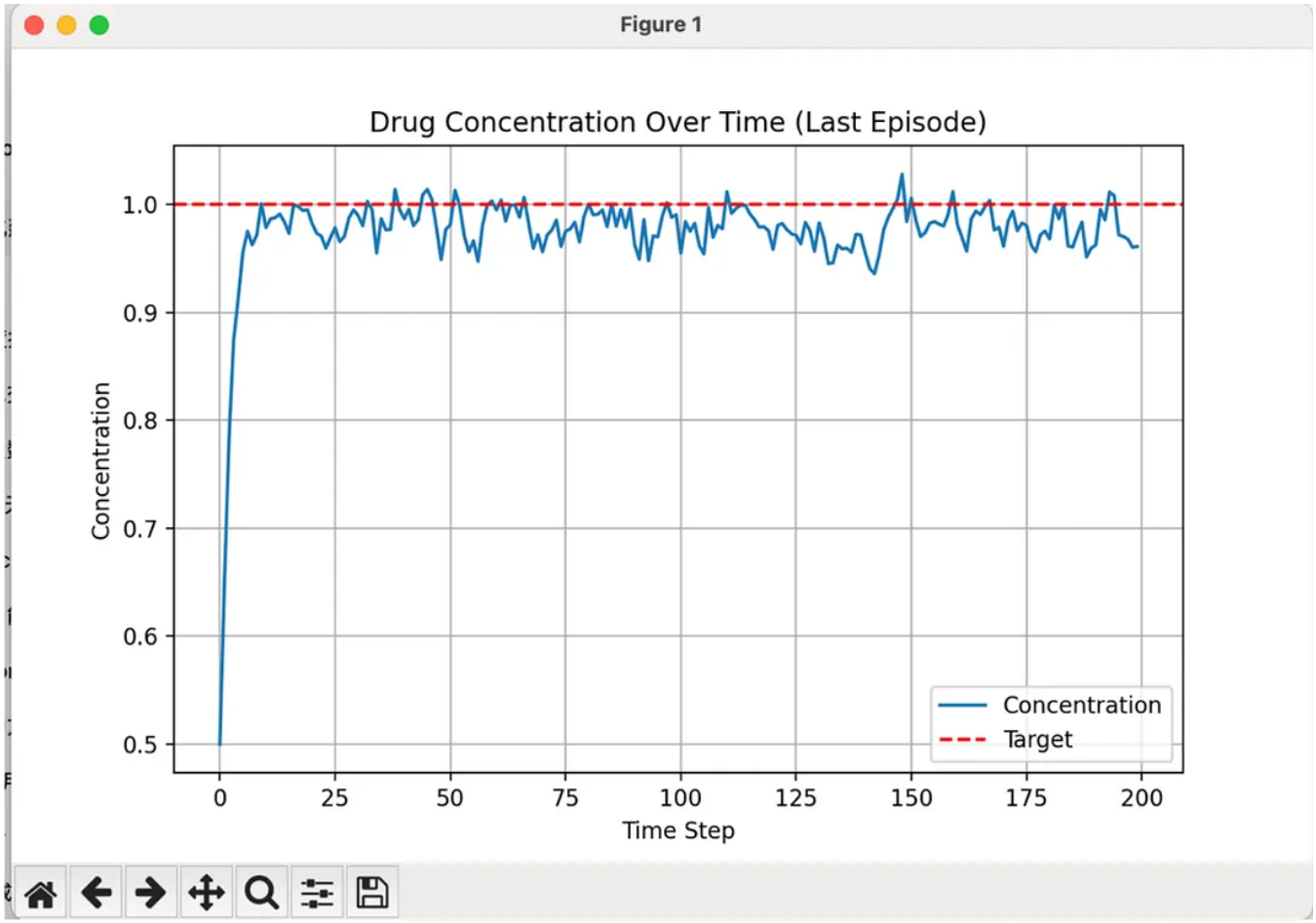


Figure 5. Drug Concentration Over Time (Last Episode, Version1)

Figure 5 presents the drug concentration trajectory in the last episode of Version1. Compared with the 1D prototype results in Figure 2, the concentration now converges to the target more quickly and exhibits smaller steady-state error. Overshoot is reduced, and the fluctuations around the target band are narrower and more symmetric. These features suggest that the updated PPO agent is not only able to reach the therapeutic level but also to maintain it in a more stable way, which is essential for safety and efficacy in a drug-dosing context.

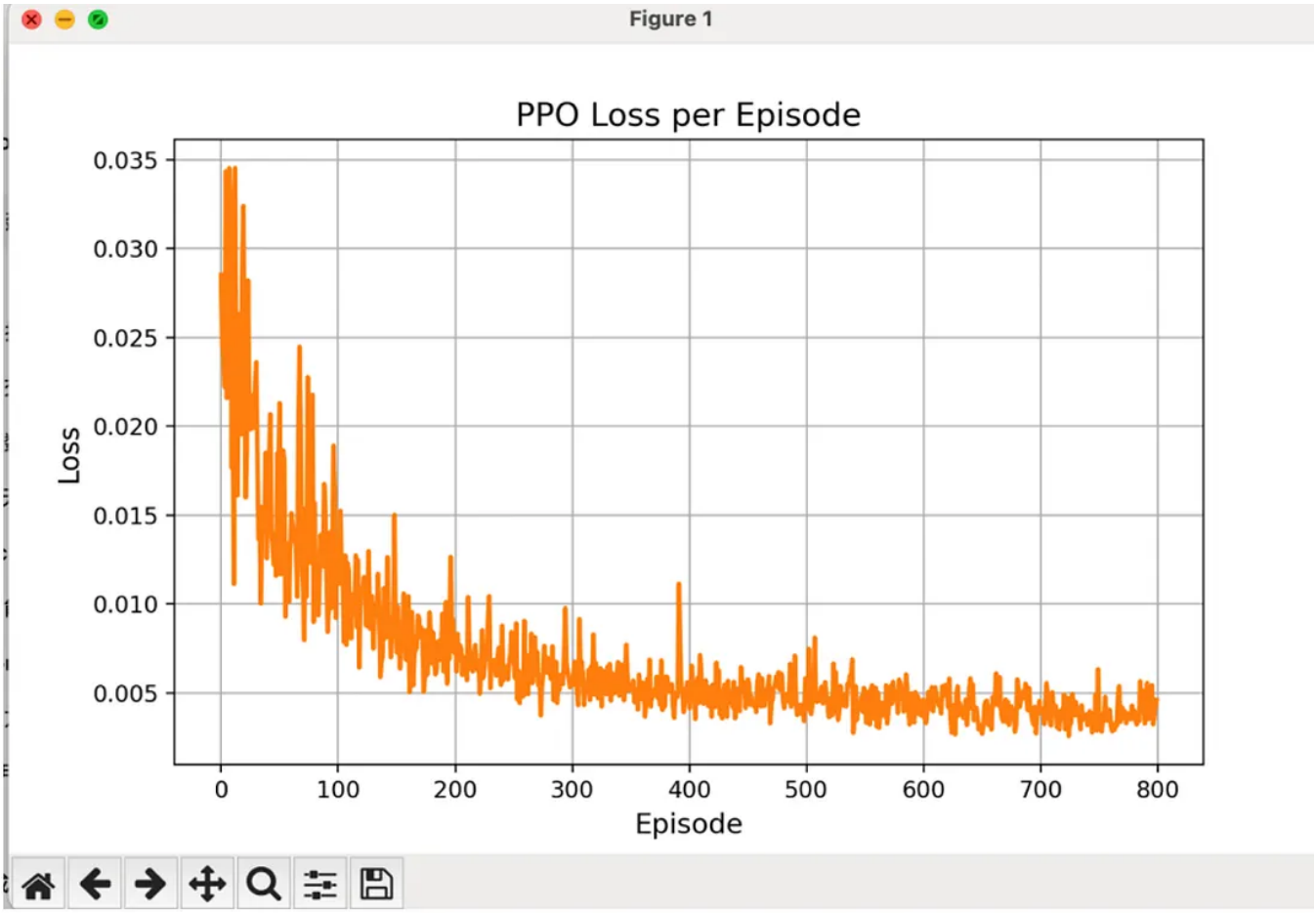


Figure 6. PPO Loss per Episode (Version1)

Figure 6 displays the PPO loss per episode after the corrections. Although the loss is still not strictly monotonic — which is typical for PPO — the overall scale is much more controlled than before, and extreme spikes are less frequent. The loss oscillates within a moderate band, indicating that the clipped objective, value loss, entropy bonus and gradient clipping are now working together coherently. This behaviour contrasts with the prototype loss in Figure 8, where large, irregular jumps reflect unstable optimisation. The stabilised loss profile in Figure 6 therefore provides additional evidence that the revised implementation leads to more reliable training dynamics.

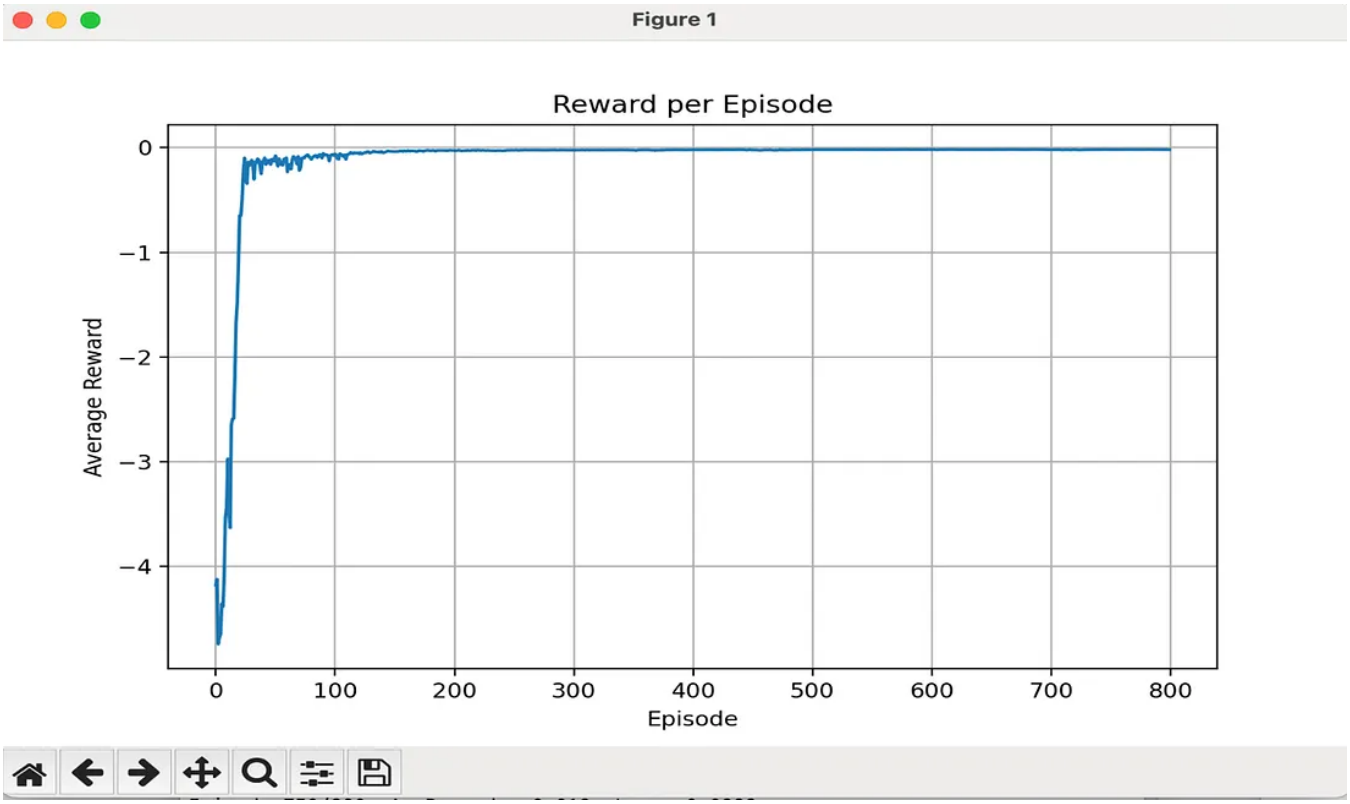


Figure 7. Reward per Episode (Original Prototype: degenerate learning)

In contrast, Figure 7 illustrates the reward progression of the original prototype. Here the curve quickly drops and then remains trapped near a low reward level, with only minor random fluctuations. This plateau corresponds to the degenerate policy that effectively chooses “no injection” across most time steps. Because the action penalty dominates early learning and the agent receives limited positive feedback for exploring, it has little incentive to deviate from this conservative behaviour. This figure therefore highlights the consequence of poorly balanced reward shaping: the agent converges to a locally safe but clinically useless strategy.

Table 1. Prototype vs Version1 — Reward Trend Comparison

Feature	Prototype (Fig. 7)	Version1 (Fig. 4)
Overall trend	Rapid drop then flat	Upward then stabilises
Convergence behaviour	Degenerate plateau	Meaningful convergence
Oscillation	Minimal, but non-informative	Moderate & informative
Learning signal clarity	Low clarity	Clearer signal
Policy implication	Zero-action tendency	Tracks target behaviour

Qualitative Conclusion: The prototype fails to learn and converges to a degenerate solution, while Version1 shows clear learning progress and stable convergence.

Table 2. Concentration Tracking — Stability & Overshoot

Feature	Prototype	Version1 (Fig. 5)
Ability to reach target	Weak	Strong
Overshoot	Higher tendency	Reduced
Steady-state error	Large	Small
Tracking stability	Poor	Improved
Variability around target	High	Low

Qualitative Conclusion: Version1 demonstrates faster convergence, reduced overshoot, and improved steady-state accuracy compared to the prototype.

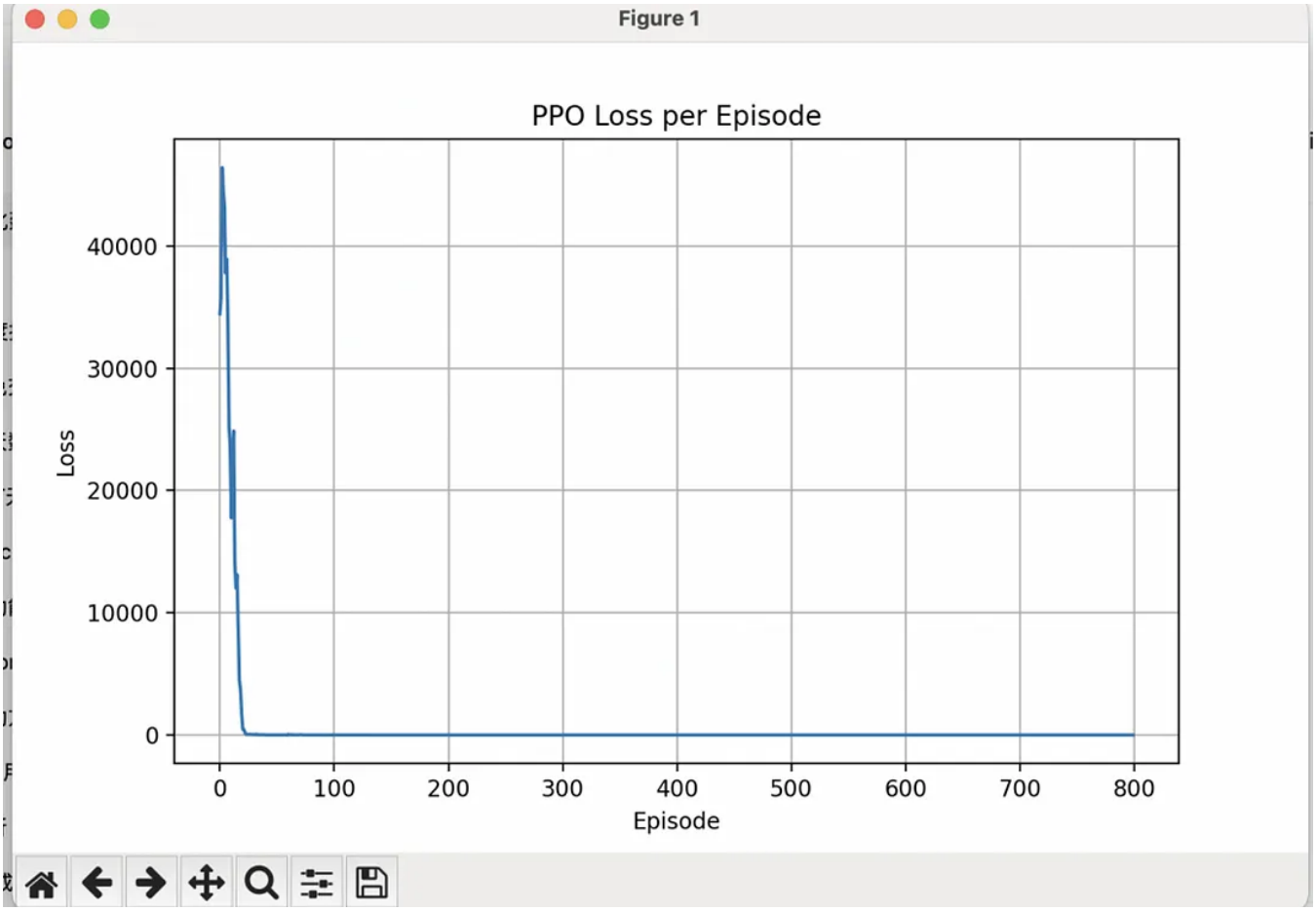


Figure 8. PPO Loss per Episode (Original Prototype)

Table 3. Loss Evolution — Qualitative Comparison

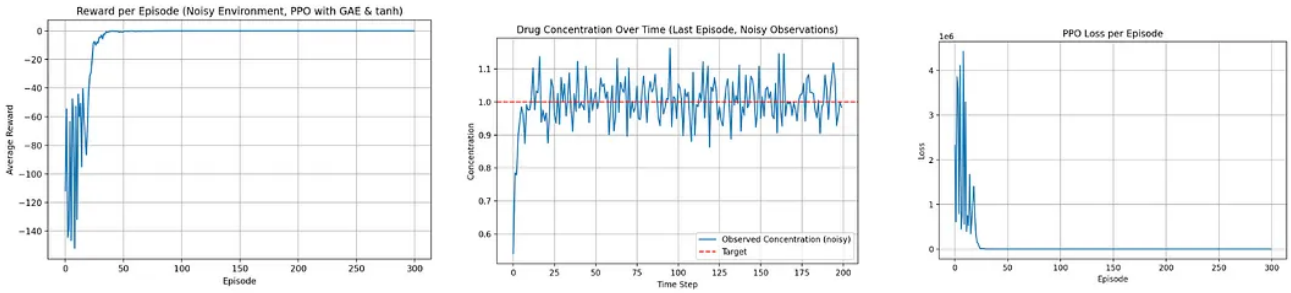
Feature	Prototype (Fig. 8)	Version1 (Fig. 6)
Spike frequency	High	Low
Loss scale stability	Poor	Controlled
Update reliability	Low	Improved
Optimisation health	Problematic	Acceptable

Qualitative Conclusion: The prototype exhibits unstable optimisation with large spikes, while Version1 shows controlled loss behaviour, indicating more reliable policy updates.

Figure 8 further confirms the instability of the prototype by showing its PPO loss per episode. The loss values exhibit sudden large jumps and irregular spikes, indicating that the policy updates are sometimes excessively aggressive or poorly aligned with the true improvement direction. These instabilities arise from inconsistent log-probability computation after action clipping, high-variance returns, and a weak

value baseline. When viewed together with Figure 7, this plot explains why the prototype fails to escape the zero-action solution: the learning signal is both noisy and misleading, preventing stable policy improvement.

4.3 Pharmacokinetic (PK) Environment



In the simplified environment, PPO converges rapidly, with the average episode reward increasing smoothly and stabilising after a small number of training episodes. The learned policy achieves accurate concentration tracking with minimal transient behaviour, and the PPO loss quickly decreases to a small, stable value, indicating well-conditioned and stable optimisation.

In contrast, the pharmacokinetic (PK) environment exhibits substantially higher initial variability. Early training is characterised by large reward fluctuations and pronounced spikes in the PPO loss, reflecting the increased difficulty introduced by multi-compartment dynamics, partial observability, and patient-to-patient variability. Despite this, PPO remains stable and eventually converges to a consistent reward level. The final concentration trajectories track the therapeutic target reliably, albeit with larger fluctuations due to hidden dynamics and stochasticity. Overall, the comparison highlights that while increased environment complexity slows convergence and amplifies training noise, PPO's stabilisation mechanisms enable robust learning without modifying the algorithm or hyperparameters.

4.4 Baseline Control Comparison (PID)

To provide a classical control reference, a PID controller is implemented as a baseline. As illustrated in Fig. 9, PID can reach the target concentration rapidly but exhibits sustained oscillations under aggressive gain tuning, highlighting its sensitivity to parameter choice and motivating the use of adaptive RL-based dosing policies.

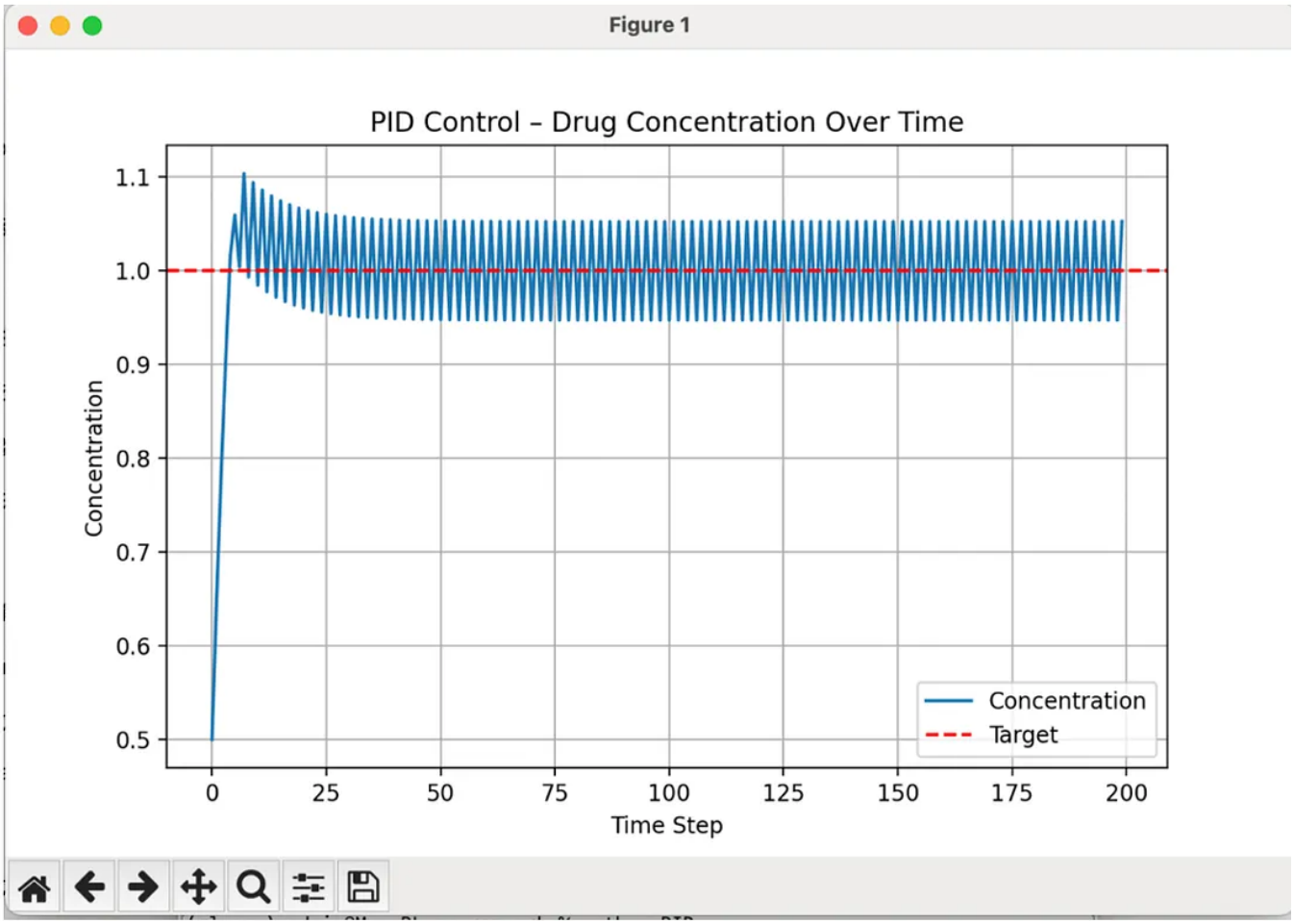


Figure 9. PID concentration tracking example (oscillation)

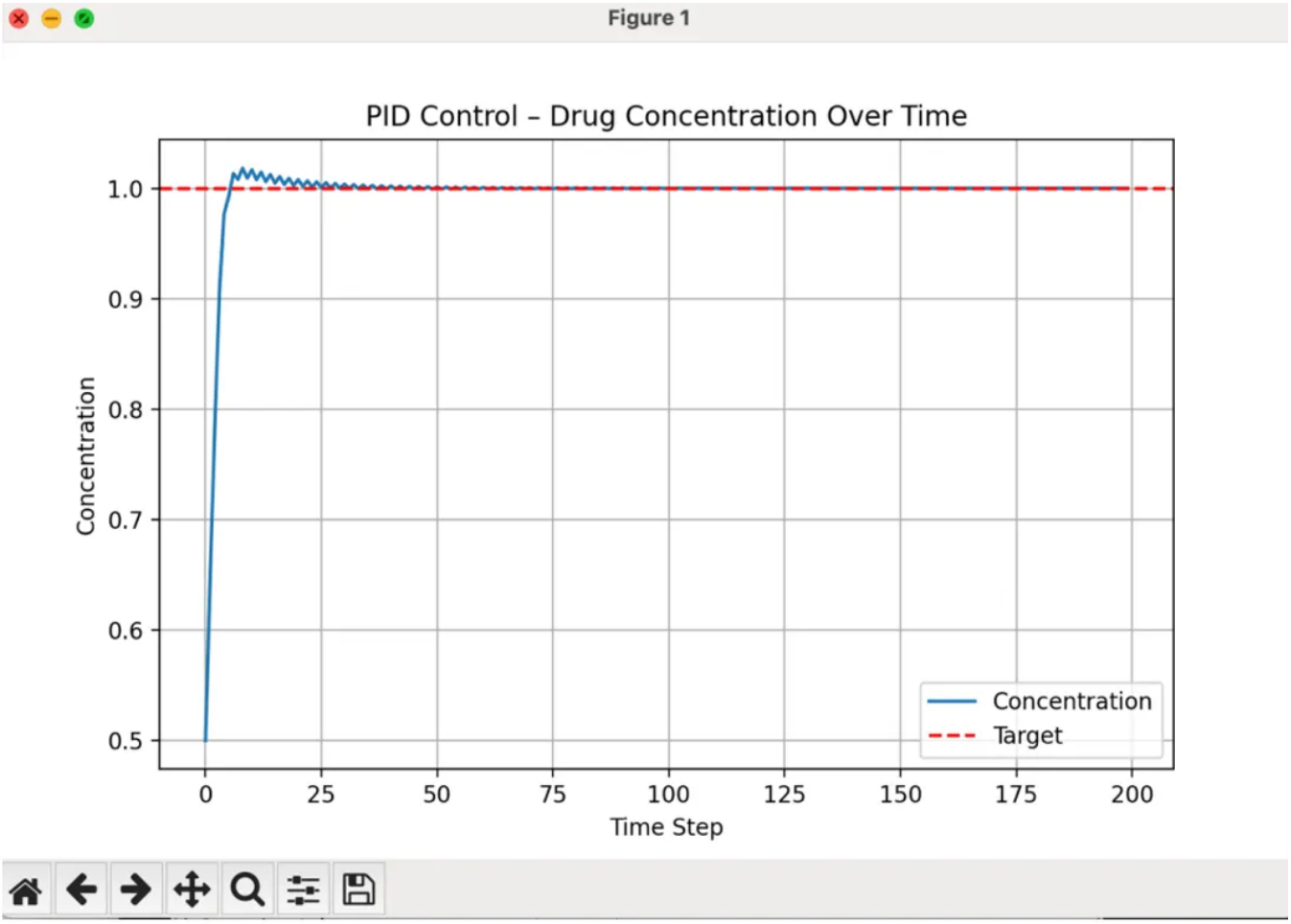


Figure 10

Figure 10 illustrates that careful PID tuning reduces oscillations and improves tracking, but this performance relies on manual gain adjustment and may not generalise across changing pharmacokinetic conditions, motivating learning-based approaches such as PPO.

Table 4. PID vs PPO — Control Behaviour Comparison

Dimension	PID (Fig. 9 & 10)	PPO (Fig. 4–6)
Oscillation	Present, especially in Fig. 9	Reduced
Overshoot	Noticeable	Reduced
Parameter dependence	High	Low
Adaptability	Low	Higher
Stability	Only after tuning	Intrinsic

Qualitative Conclusion: PID can achieve good performance but requires manual tuning and is sensitive to

conditions, whereas PPO learns adaptively and offers more robust control.

5. Bioengineering Contextualisation

Reinforcement learning, particularly PPO, has great potential in bioengineering applications that require adaptive, real-time control. In this tutorial, we demonstrated PPO for regulating blood drug concentration in a simplified digital environment. This approach can be extended to real-world personalized drug delivery systems, where the goal is to maintain therapeutic drug levels within a patient's bloodstream while minimizing side effects. For instance, an RL agent could continuously adjust infusion rates in response to real-time sensor measurements, adapting to individual patient metabolism, circadian variations, or other physiological changes.

Beyond drug delivery, PPO and other RL methods could also be applied to robotic prosthetics: Adaptive control for movement assistance based on real-time biomechanical feedback. Rehabilitation robotics: Adjust exercise intensity or support in response to patient progress. Synthetic biology: Regulate gene expression or metabolic pathways in microbial cultures for optimized production. In summary, the PPO algorithm's ability to learn robust policies in continuous, dynamic environments makes it a promising tool for adaptive, personalized bioengineering solutions that require real-time decision-making.

I acknowledge the use of ChatGPT 5 (OpenAI, https://chat.openai.com/) to generate an outline for background study. I confirm that no content generated by AI has been presented as my own work.

6. References

[1] Wang, X., Ma, Z., Cao, L. et al. A planar tracking strategy based on multiple-interpretable improved PPO algorithm with few-shot technique. Sci Rep 14, 3910 (2024). https://doi.org/10.1038/s41598-024-54268-6

[2] de Miguel Gomez, A. and Toosi, F. G. (2021). Continuous Parameter Control in Genetic Algorithms using Policy Gradient Reinforcement Learning. In Proceedings of the 13th International Joint Conference on Computational Intelligence (IJCCI 2021) — ECTA; ISBN 978–989–758–534–0; ISSN 2184–3236, SciTePress, pages 115–122. DOI: 10.5220/0010643500003063

[3] Kargin, T.C., Kołota, J. A Reinforcement Learning Approach for Continuum Robot Control. J Intell Robot Syst 109, 77 (2023). https://doi.org/10.1007/s10846-023-02003-0

[4] D. Adamu Aliyu, E. Akashah Patah Akhir, M. Omar Abdullah Sawad, J. Shehu Yalli and Y. Saidu, "A Reinforcement Learning Approach to Personalized Asthma Exacerbation Prediction Using Proximal Policy Optimization," in IEEE Access, vol. 13, pp. 103373–103384, 2025, doi: 10.1109/ACCESS.2025.3579198.

[5] C. -H. Cho, P. -J. Huang, M. -C. Chen and C. -W. Lin, "Closed-Loop Deep Brain Stimulation With Reinforcement Learning and Neural Simulation," in IEEE Transactions on Neural Systems and Rehabilitation Engineering, vol. 32, pp. 3615–3624, 2024, doi: 10.1109/TNSRE.2024.3465243.

[6] D. F. G. Filho, M. A. Moret and E. G. S. Nascimento, "A Custom Reinforcement Learning Environment for Hybrid Renewable Energy Systems: Design and Implementation," in IEEE Access, vol. 13, pp. 133984–133993, 2025, doi: 10.1109/ACCESS.2025.3593064.

[7] Tan, C.; Wang, J.; Cai, H.; Hu, S.; Zhang, B.; Zhu, W. Phase-Adaptive Reinforcement Learning for Self-Tuning PID Control of Cruise Missiles. Aerospace 2025, 12, 849. https://doi.org/10.3390/aerospace12090849

[8] Jifei Deng, Seppo Sierla, Jie Sun, Valeriy Vyatkin, Reinforcement learning for industrial process control: A case study in flatness control in steel industry, Computers in Industry, 143, 2022, 103748, ISSN 0166–3615, https://doi.org/10.1016/j.compind.2022.103748.

[9] Garcia, C. (2024). Reinforcement learning for dynamic pricing and capacity allocation in monetized customer wait-skipping services. Journal of Business

Analytics, 8(1), 36–54. https://doi.org/10.1080/2573234X.2024.2424542

[10] F. Rahimi Ghashghaei, N. Elmrabit, A. -U. -H. Qureshi, A. Akhunzada and M. Yousefi, “Advanced Quantum Control With Ensemble Reinforcement Learning: A Case Study on the XY Spin Chain,” in IEEE Access, vol. 13, pp. 49514–49526, 2025, doi: 10.1109/ACCESS.2025.3551232

Edit profile

Written by Sheng Jiang

0 followers · 1 following

No responses yet

Sheng Jiang

What are your thoughts?

Recommended from Medium

Jae Geun Lee (Neuroscientist & Frame Jump Theory)

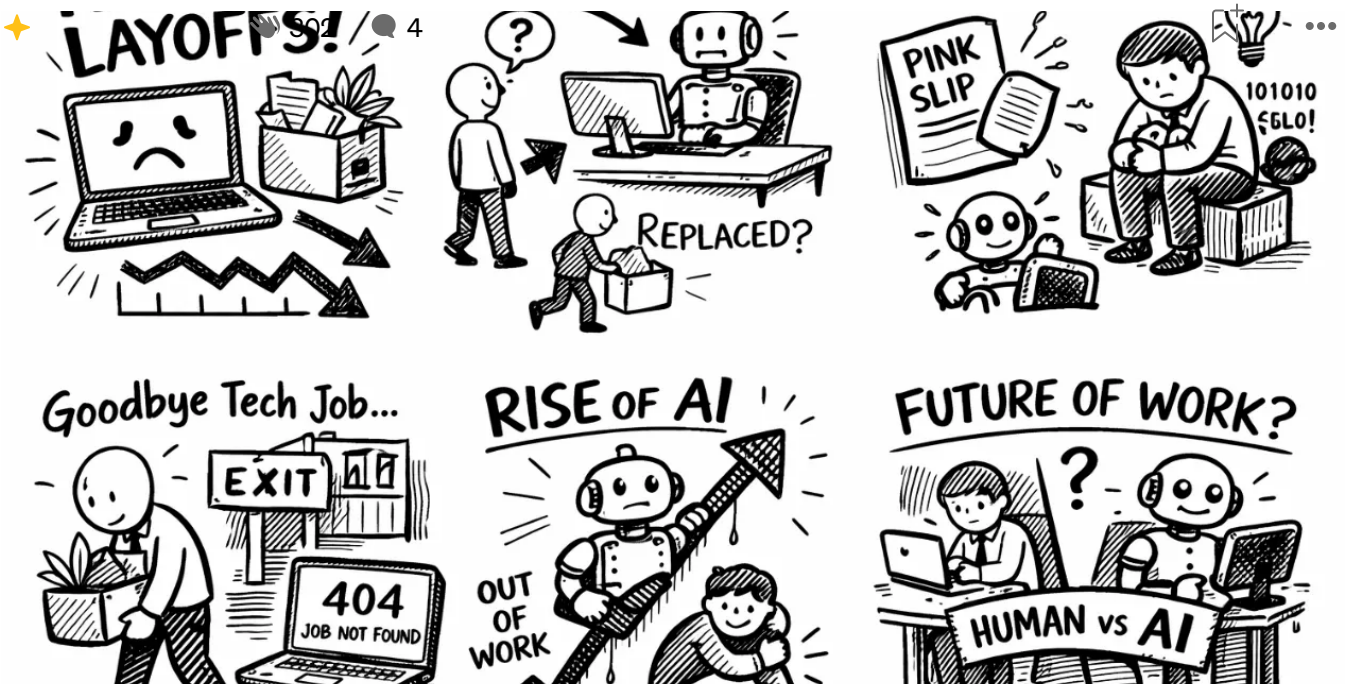
🎓Why Graduate School Was Built for Critical Thinking—But No One's Allowed to Think Critically

Oct 23, 2025

In Towards AI by Adham Khaled

The Constructive Lie: Why Telling Your LLM the Wrong Answer Makes It Smarter

Stop asking your AI to “think step-by-step.” Start asking it, “Why is this wrong?”

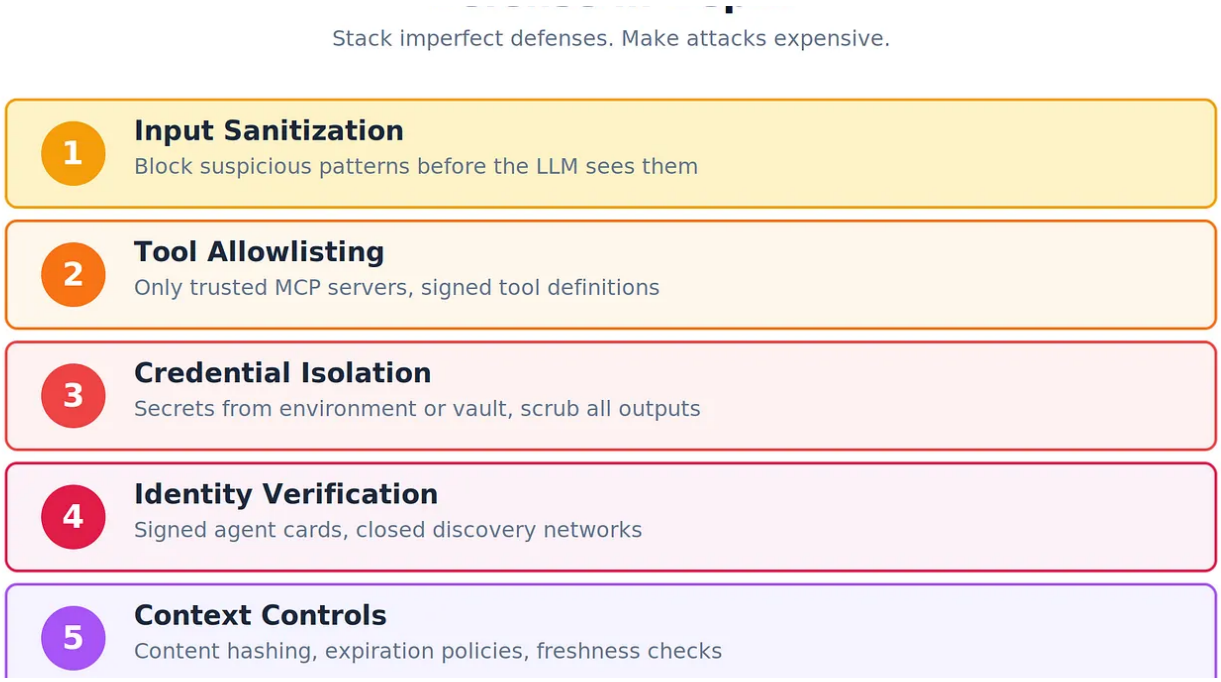


In Write A Catalyst by Adarsh Gupta

An Honest Recruiter Told Me Why ‘Average Developers’ Are Slowly Disappearing...

It’s already too late for people who haven’t realized it.

Jan 2 1.1K 45



In Data Science Collective by Paolo Perrone

5 Ways Your AI Agent Will Get Hacked (And How to Stop Each One)

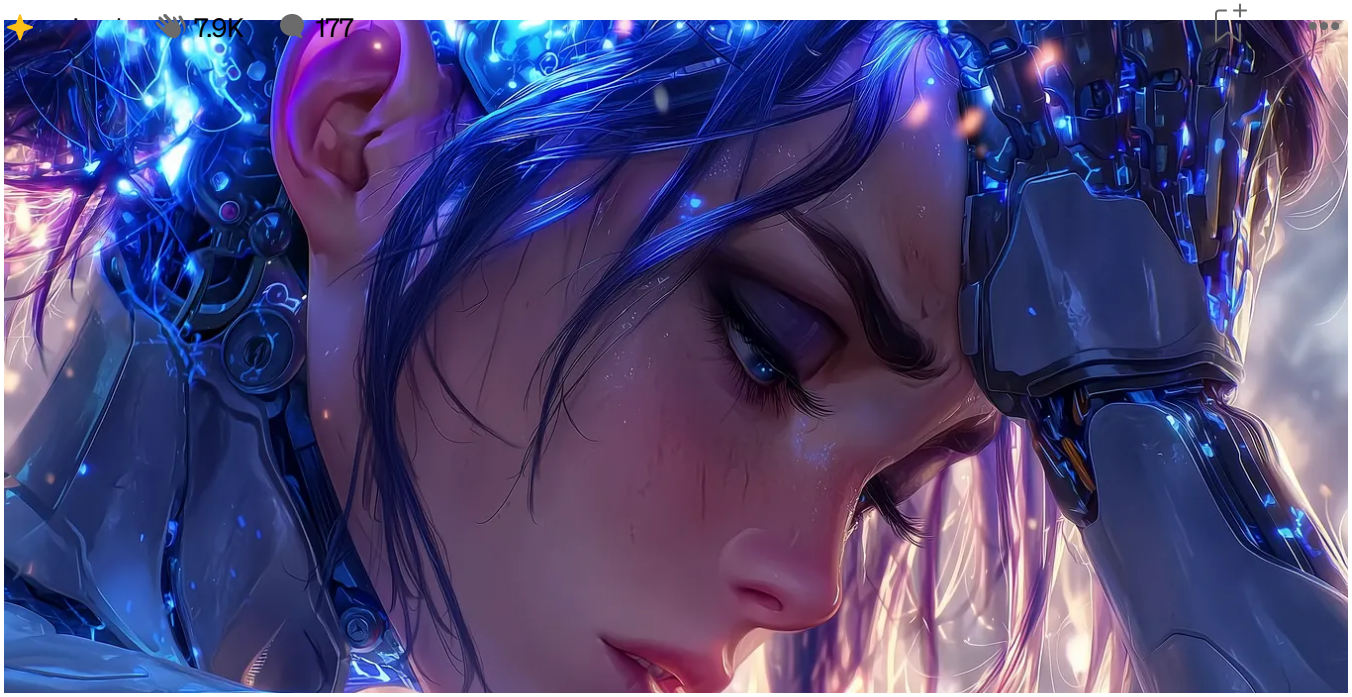
Prompt injection is just the beginning

5d ago 286 7

In Publishous by Joshua Mason

I'm an Interrogator, and These "Harmless" Conversation Habits Are Actually Red Flags

Sometimes you just need to head for the exit—fast



In AI Advances by Delanoe Pirard

The AI Learned to Think on Its Own. Nobody Taught It How.

In January 2025, a Chinese lab taught an AI to reason by itself. 97% of the time, that AI now pretends to obey us while secretly preserving...

5d ago 366 8

See more recommendations