Beyond Scalar Rewards: Dense Feedback for LLM Policy Synthesis in Sequential Social Dilemmas

author names withheld

Under Review for NExT-Game 2026

Abstract

We study *LLM policy synthesis*: using a language model to iteratively generate programmatic agent policies for multi-agent environments. Rather than training neural policies via reinforcement learning, our framework prompts an LLM to produce Python policy functions, evaluates them in self-play, and refines them using performance feedback across iterations. We investigate *feedback engineering* (the design of what evaluation information is shown to the LLM during refinement) comparing *sparse feedback* (scalar reward only) against *dense feedback* (reward plus social metrics: efficiency, equality, sustainability, peace). Across two canonical Sequential Social Dilemmas (Gathering and Cleanup) and two frontier LLMs (Claude Sonnet 4.6, Gemini 3.1 Pro), dense feedback consistently matches or exceeds sparse feedback on all metrics. We explain the asymmetry through *feedback aliasing*: when scalar reward alone maps distinct failure modes to the same value (e.g., under- vs. over-cleaning), social metrics break the alias and let the LLM diagnose which corrective direction to take. Social metrics thus function as a *coordination signal* rather than a distraction, yielding strategies such as Voronoi territory partitioning and waste-adaptive cleaner schedules.
Code at https://github.com/anon590/llm-policies-social-dilemmas.

1. Introduction

Sequential Social Dilemmas (SSDs) [6] are multi-agent environments where individually rational behavior leads to collectively suboptimal outcomes, e.g., they are the multi-agent analog of the prisoner's dilemma, extended to temporally rich Markov games. Standard multi-agent reinforcement learning (MARL) struggles with SSDs due to credit assignment difficulties, non-stationarity, and the vast joint action space [2].

Recent advances in large language models (LLMs) open a fundamentally different approach: rather than learning policies through gradient-based optimization in *parameter space*, an LLM can directly *synthesize programmatic policies* in *algorithm space*: writing executable code that implements complex coordination strategies such as territory division, role assignment, and conditional cooperation. This paradigm, related to FunSearch [12] and Eureka [8], sidesteps the sample efficiency bottleneck of MARL entirely: a single LLM generation step can produce a sophisticated coordination algorithm that would require millions of RL episodes to discover.

A critical question arises when using iterative LLM synthesis: *what feedback should the LLM receive between iterations?* We introduce *feedback engineering* as a design axis and compare *sparse* (scalar reward only) against *dense* feedback (reward plus efficiency, equality, sustainability, and peace). Across two frontier LLMs (Claude Sonnet 4.6, Gemini 3.1 Pro) and two canonical SSDs (Gathering, Cleanup), dense feedback consistently matches or exceeds sparse feedback on all metrics,

with a 54% efficiency gain in Cleanup. Across both games and both LLMs, dense feedback also produces higher equality and sustainability without sacrificing efficiency. We explain this asymmetry through *feedback aliasing*: when distinct failure modes (under- vs. over-cleaning) collapse to the same scalar reward, social metrics break the alias and indicate the corrective direction. When no such alias exists (Gathering), both modes converge to similar performance.

2. Framework

2.1. Sequential Social Dilemmas

An SSD is a partially observable Markov game $\mathcal{G} = \langle N, \mathcal{S}, \{\mathcal{A}_i\}_{i=1}^N, T, \{R_i\}_{i=1}^N, H\rangle$ with N agents, state space $\mathcal{S}$ (the gridworld configuration), per-agent action spaces $\mathcal{A}_i$, transition function T, reward functions R_i, and episode horizon H. We study two canonical SSDs from the literature.

Gathering [6]. Agents navigate a 2D gridworld and collect apples ($+1$ reward). Apples respawn on a fixed 25-step timer. Agents may fire a tagging beam (2 hits remove a rival for 25 steps). The dilemma: agents can coexist peacefully and share resources, or attack rivals to monopolize apples, but aggression wastes time and reduces total welfare.

Cleanup [5]. A public goods game with two regions: a river that accumulates waste, and an orchard where apples grow. Apples only regrow when the river is sufficiently clean. Agents can fire a cleaning beam (costs -1) to remove waste, or collect apples ($+1$). A penalty beam (costs -1, inflicts -50 on the target) can tag rivals out for 25 steps. The dilemma: cleaning is costly but benefits everyone; purely selfish agents free-ride on others' cleaning.

Both games use 8–9 discrete actions (4 movement directions, 2 rotations, beam, stand, and optionally clean) and episodes of $H = 1000$ steps. Screenshots of both environments are shown in Figure 4 (Appendix).

Following Perolat et al. [11], we evaluate outcomes using four social metrics. Let $R_i = \sum_{t=0}^{H-1} r_i^t$ denote agent i's episode return. Then:

$$\text{Efficiency:} \quad U = \frac{1}{H} \sum_{i=1}^{N} R_i \tag{1}$$

$$\text{Equality:} \quad E = 1 - \frac{\sum_{i,j} |R_i - R_j|}{2N \sum_i R_i} \tag{2}$$

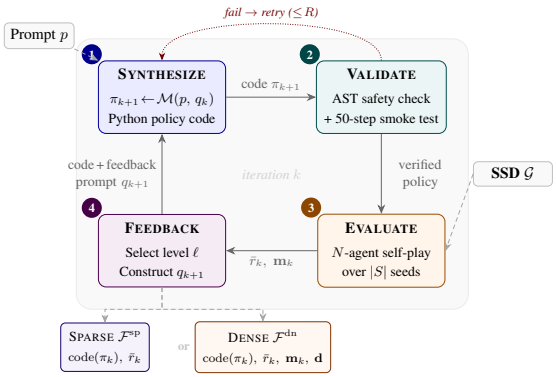
$$\text{Sustainability:} \quad S = \frac{1}{N} \sum_{i=1}^{N} \bar{t}_i \tag{3}$$

$$\text{Peace:} \quad P = \frac{1}{H} \sum_{t=0}^{H-1} \left|\{i : \text{active}_i^t\}\right| \tag{4}$$

where $\bar{t}_i$ is the mean timestep at which agent i collects positive reward (higher means resources remain available later), and active_i^t indicates agent i is not tagged out at step t.

2.2. Iterative LLM Policy Synthesis

Let Π denote the space of *programmatic policies*: deterministic functions $\pi : \mathcal{S} \times [N] \to \mathcal{A}$ expressed as executable Python code. Each policy has access to the full environment state and a library of helper functions: breadth-first search (BFS) pathfinding, beam targeting, and coordinate transforms. This state access is a deliberate design choice: programmatic policies operate in algorithm space rather than in the reactive observation-to-action space of neural policies. Code as Policies [7]



Algorithm 1: Iterative LLM Policy Synthesis

Input: Game $\mathcal{G}$, LLM $\mathcal{M}$, prompt p,
iterations K, level ℓ, seeds S
Output: Final policy π_K

$\pi_0 \leftarrow \mathcal{M}(p,$ "generate initial policy")
$\mathcal{F}_0^\ell \leftarrow \text{Eval}(\pi_0;\ \mathcal{G},\ S)$
for $k = 1, \ldots, K$ **do**
  for attempt $= 1, \ldots, R$ **do**
    $\pi_k \leftarrow \mathcal{M}\big(p,\ q(\pi_{k-1}, \mathcal{F}_{k-1}^\ell)\big)$ **if** Validate(π_k) **then break**
    $q \leftarrow q \,\|\, \text{error\_msg}$
  end
  $\mathcal{F}_k^\ell \leftarrow \text{Eval}(\pi_k;\ \mathcal{G},\ S)$
end
return π_K

Figure 1: **Iterative LLM policy synthesis.** *Left:* the four-phase cycle. At each iteration k, the LLM SYNTHESIZES (**1**) a Python policy from the system prompt p and previous feedback, which is VALIDATED (**2**) via AST checks and a smoke test (retrying on failure up to R times), EVALUATED (**3**) in N-agent self-play, and the results packaged as either *sparse* or *dense* FEEDBACK (**4**). *Right:* the procedure in pseudocode.

demonstrated that LLMs can generate executable robot policy code that processes perception outputs and parameterizes control primitives via few-shot prompting. And Eureka [8] uses LLMs to generate code-based reward functions (rather than policies) from the environment source code. Our work differs from these in that the LLM iteratively synthesizes complete agent policies for a multi-agent setting, where the generated code must simultaneously coordinate across agents sharing the same program.

A frozen LLM $\mathcal{M}$ acts as a *policy synthesizer*. Given a system prompt p describing the environment API and a feedback prompt q_k, it generates source code implementing a new policy $\pi_{k+1} = \mathcal{M}\big(p,\ q\big(\pi_k,\ \mathcal{F}_k^\ell\big)\big)$, where π_k is the previous policy (its source code), $\mathcal{F}_k^\ell$ is the evaluation feedback at level ℓ, and $q(\cdot)$ constructs the user prompt. All N agents execute the same policy π_k (homogeneous self-play). The evaluation computes feedback over a set of random seeds S:

$$\mathcal{F}_k = \text{Eval}(\underbrace{\pi_k, \ldots, \pi_k}_{N};\ \mathcal{G},\ S) = (\bar{r}_k,\ \mathbf{m}_k)$$

where $\bar{r}_k = \frac{1}{N|S|} \sum_{s \in S} \sum_i R_i^{(s)}$ is the mean per-agent return and $\mathbf{m}_k = (U_k, E_k, S_k, P_k)$ is the social metrics vector.

Each generated policy undergoes AST-based safety checking (blocking dangerous operations such as `eval`, file I/O, and network access) followed by a 50-step smoke test to catch runtime errors. If validation fails, the error message is appended to the prompt and generation is retried (up to 3 attempts).

2.3. Feedback Engineering

We define two feedback levels ℓ that control what information the LLM receives between iterations:

Sparse feedback (REWARD-ONLY). The LLM receives the previous policy's source code and the scalar mean per-agent reward:

$$\mathcal{F}_k^{\mathrm{sp}} = (\,\mathrm{code}(\pi_k),\ \bar{r}_k\,) \tag{5}$$

Dense feedback (REWARD+SOCIAL). The LLM additionally receives the full social metrics vector together with natural-language definitions of each metric:

$$\mathcal{F}_k^{\mathrm{dn}} = (\,\mathrm{code}(\pi_k),\ \bar{r}_k,\ \mathbf{m}_k,\ \mathbf{d}\,) \tag{6}$$

where $\mathbf{d}$ contains textual definitions (e.g., "*Equality: fairness of reward distribution, 1.0 = perfectly equal*"). We avoid leaking environment information in these definitions, to ensure a fair comparison between methods.

In both modes, the system prompt instructs the LLM to *maximize per-agent reward*: the social metrics in dense feedback are presented as informational context, not explicit optimization targets. Both modes use the neutral framing "all agents run the same code" (no adversarial language, nor placing emphasis on cooperation). At iteration 0 the LLM generates a policy from scratch; subsequent iterations receive the previous policy's code together with its feedback. The crucial design question is whether $\ell = \mathrm{sp}$ or $\ell = \mathrm{dn}$ produces better policies.

3. Experiments

3.1. Setup

We run both SSDs with $N = 10$ agents on large-map variants, $K = 3$ refinement iterations, $|S| = 5$ evaluation seeds, and 3 independent runs per configuration. We evaluate two frontier LLMs, **Claude Sonnet 4.6** and **Gemini 3.1 Pro**, both with maximum thinking budget. For each model $\times$ game, we compare three settings: ZERO-SHOT (no refinement), REWARD-ONLY (sparse feedback), and REWARD+SOCIAL (dense feedback). As other baselines, we run a tabular **Q-learner** with hand-crafted features and cooperative reward shaping, a hand-coded **BFS Collector** (nearest-apple navigation, no beaming), and **GEPA** [1], an LLM-based prompt optimizer using the same Gemini 3.1 Pro and matched compute budget; GEPA's reflection LM receives only scalar reward, matching REWARD-ONLY's information. Full details are in Appendix B.

3.2. Main Results

Table 1 presents results across both games, both models, and all feedback configurations. Three findings emerge.

Finding 1: LLM policy synthesis dominates traditional and prompt-level baselines. All refined LLM policies dramatically outperform non-LLM baselines. In Gathering, the best configuration (Gemini, dense, $U = 4.59$) achieves $6.0\times$ the Q-learner ($U = 0.77$) and $3.6\times$ the BFS heuristic ($U = 1.29$); in Cleanup, $U = 2.75$ vs. -0.16, as tabular Q-learning fails at the cleaning–harvesting credit assignment. Iterative refinement is important: Claude's ZERO-SHOT scores $U = -1.01$ in Cleanup, rising to $U = 1.14$–1.37 after 3 iterations. GEPA, which optimizes the system prompt rather than the code, trails direct code-level iteration by 25% in Gathering ($U = 3.45$ vs. 4.59) and $3.6\times$ in Cleanup ($U = 0.77$ vs. 2.75), with highly negative equality ($E = -1.75$) indicating free-riding: code-level feedback is substantially more effective than prompt-level meta-optimization for cooperative strategy discovery.

Table 1: Results across two SSDs, two LLMs, and three feedback configurations. LLM values show the mean over 3×5 independent runs (min–max in parentheses). U: efficiency (Eq. 1). E: equality (Eq. 2). S: sustainability (Eq. 3). Bold marks the best value per game $\times$ model block. Baselines (bottom of each game block) are non-LLM methods for reference.

Game	Model	Feedback	U	E	S
Gathering	Claude Sonnet 4.6	ZERO-SHOT	1.85 (1.52–2.02)	0.52 (0.35–0.63)	298.6 (180–374)
		REWARD-ONLY	3.47 (3.39–3.56)	0.72 (0.61–0.87)	402.9 (314–500)
		REWARD+SOCIAL	**3.53** (1.60–4.58)	**0.84** (0.63–0.94)	**452.7** (356–502)
	Gemini 3.1 Pro	ZERO-SHOT	3.71 (1.90–4.46)	0.79 (0.35–0.96)	443.2 (180–504)
		REWARD-ONLY	4.58 (4.48–4.63)	**0.97** (0.97–0.97)	502.5 (502–503)
		REWARD+SOCIAL	**4.59** (4.50–4.65)	**0.97** (0.97–0.97)	**502.7** (501–505)
	GEPA (Gemini 3.1 Pro)		3.45 (3.16–3.95)	0.91 (0.83–0.96)	496.2 (491–499)
	Q-learner		0.77 (0.73–0.81)	0.83 (0.81–0.85)	508.2 (504–513)
	BFS Collector		1.29	0.54	489.5
Cleanup	Claude Sonnet 4.6	ZERO-SHOT	−1.01 (−7.90–0.93)	−3.06 (−15.9–1.00)	137.0 (19–214)
		REWARD-ONLY	1.14 (1.04–1.32)	−0.47 (−0.66–−0.11)	233.0 (225–245)
		REWARD+SOCIAL	**1.37** (1.16–1.73)	**0.09** (−0.15–0.55)	**294.6** (209–456)
	Gemini 3.1 Pro	ZERO-SHOT	0.45 (−2.32–1.28)	−0.45 (−1.38–0.73)	274.1 (158–421)
		REWARD-ONLY	1.79 (1.16–2.57)	0.13 (−0.47–0.47)	386.0 (186–503)
		REWARD+SOCIAL	**2.75** (1.44–3.49)	**0.54** (0.33–0.67)	**432.6** (297–501)
	GEPA (Gemini 3.1 Pro)		0.77 (0.58–1.07)	−1.75 (−2.54–−0.62)	209.5 (184–260)
	Q-learner		−0.16 (−0.28–−0.06)	0.20 (−0.37–1.00)	208.6 (121–301)
	BFS Collector		0.10	0.61	16.4

Finding 2: Dense feedback consistently matches or exceeds sparse feedback. Across all four game $\times$ model combinations, REWARD+SOCIAL achieves equal or higher efficiency than REWARD-ONLY. The advantage is most pronounced in Cleanup: Gemini gains 54% (U: 2.75 vs. 1.79) and Claude 20% (1.37 vs. 1.14). In Gathering the modes perform similarly, with dense feedback holding a slight edge for Claude (3.53 vs. 3.47).

Finding 3: Social metrics serve as a coordination signal. Dense feedback simultaneously improves efficiency, equality, and sustainability without tradeoffs (e.g., Cleanup/Gemini: E rises from 0.13 to 0.54, S from 386 to 433 while U also peaks). Inspecting the generated code (Appendix C) reveals how: under dense feedback, the LLM writes *waste-adaptive cleaner schedules* that scale the cleaning team with pollution level and *BFS-Voronoi territory partitioning* with zero aggression in Gathering. Sparse-feedback policies instead use fixed cleaning roles and multi-tier combat systems that waste actions. Between models, Gemini 3.1 Pro is consistently stronger and lower-variance than Claude Sonnet 4.6 (e.g., Gathering/dense: $U \in [4.50, 4.65]$ vs. $[1.60, 4.58]$), with the largest gap in Cleanup (2.75 vs. 1.37).

3.3. Why Dense Feedback Helps: Feedback Aliasing

The asymmetry in Table 1 (dense feedback's efficiency gain is large in Cleanup but negligible in Gathering) admits a simple explanation we call *feedback aliasing*: whether scalar reward alone uniquely identifies the correct refinement direction. In Cleanup, total reward is a concave function

of the number of cleaners n_c with an interior optimum. Below this peak, *two* distinct failure modes produce the same scalar reward but require opposite corrections: under-cleaning (too few cleaners, apples starved by pollution) and over-cleaning (too many cleaners, cost exceeds marginal benefit). Social metrics break the alias: under-cleaning manifests as low sustainability, over-cleaning as low equality (cleaners bear the cost, harvesters capture the reward). Dense feedback thus gives the LLM a diagnostic signal that sparse feedback lacks, and this is visible in the code: under dense feedback the LLM writes waste-adaptive schedules that scale n_c with pollution (Appendix C.2), while sparse feedback produces fixed allocations that cannot self-correct. Gathering has no such alias: the coordination problem reduces to reducing territorial overlap on a single axis. Hence, the scalar reward already indicates the corrective direction, and both modes converge.

4. Related Work

Sequential Social Dilemmas. SSDs were introduced by Leibo et al. [6] (Gathering) and extended to public goods settings by Hughes et al. [5] (Cleanup); Perolat et al. [11] formalized the social outcome metrics we use.

LLMs for policy and program synthesis. FunSearch [12] evolves programs for combinatorial problems; Eureka [8] synthesizes reward functions; Voyager [16] and Code as Policies [7] generate executable skill code for single-agent control; ReEvo [18] evolves heuristics via reflection. We target *multi-agent* policy code where a single program must coordinate across agents.

LLM reflection and feedback. Reflexion [14], Self-Refine [9], OPRO [17], and GEPA [1] all demonstrate that structured verbal feedback loops improve LLM outputs; ERL [13] internalizes such reflection via self-distillation. We complement this line by varying the *content* of feedback (scalar vs. multi-objective social metrics) in a multi-agent setting.
An extended version of this section is located at Appendix E.

5. Discussion and Conclusion

Richer feedback helps: dense social metrics consistently match or exceed sparse scalar reward across two games and two frontier LLMs. Rather than triggering over-optimization of fairness, social metrics act as a coordination signal that disambiguates distinct failure modes collapsed by scalar reward (the feedback aliasing argument of Section 3.3). This work extends the LLM-reflection literature [9, 14] to multi-agent settings where the feedback dimensions capture social outcomes.

Limitations and future work. Our SSDs are small-scale; scaling to larger environments, heterogeneous per-agent policies, and intermediate feedback levels (e.g., partial social metrics) are natural next steps. On the safety side, an open question is whether exploits emerge organically during iterative synthesis without explicit adversarial prompting. While in the previous experiments we didn't find evidence of the LLM reward hacking the environment, we separately examined whether LLMs can reward-hack our framework by mutating the environment object they receive. An adversarially prompted Claude Opus 4.6 autonomously discovered five distinct attacks (Appendix F), with dynamics-bypass attacks amplifying reward up to $59\times$ while simultaneously improving measured social metrics, a Goodharting concern with direct implications for verification pipelines in LLM policy synthesis.

References

[1] Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. Gepa: Reflective prompt evolution can outperform reinforcement learning, 2026. URL https://arxiv.org/abs/2507.19457.

[2] Lucian Busoniu, Robert Babuska, and Bart De Schutter. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 38(2):156–172, 2008. doi: 10.1109/TSMCC.2007.913919.

[3] Víctor Gallego. Specification self-correction: Mitigating in-context reward hacking through test-time refinement. In *The 1st Workshop on Test-time Scaling and Reasoning Models*, 2025. URL https://openreview.net/forum?id=UU9KCA0sTH.

[4] Charles AE Goodhart. Problems of monetary management: The UK experience. *Monetary Theory and Practice*, pages 91–121, 1984.

[5] Edward Hughes, Joel Z. Leibo, Matthew Phillips, Karl Tuyls, Edgar A. Duéñez-Guzmán, Antonio García Castañeda, Iain Dunning, Tina Zhu, Kevin R. McKee, R. Koster, Heather Roff, and Thore Graepel. Inequity aversion improves cooperation in intertemporal social dilemmas. In *Neural Information Processing Systems*, 2018. URL https://api.semanticscholar.org/CorpusID:52843325.

[6] Joel Z Leibo, Vinicius Zambaldi, Marc Lanctot, Janusz Janssen, and Thore Graepel. Multi-agent reinforcement learning in sequential social dilemmas. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, pages 464–473, 2017.

[7] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *2023 IEEE International Conference on Robotics and Automation (ICRA)*, pages 9493–9500, 2023. doi: 10.1109/ICRA48891.2023.10160591.

[8] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-level reward design via coding large language models. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=IEduRUO55F.

[9] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=S37hOerQLB.

[10] Alexander Pan, Kush Bhatia, and Jacob Steinhardt. The effects of reward misspecification: Mapping and mitigating misaligned models. In *International Conference on Learning Representations*, 2022.

[11] Julien Perolat, Joel Z Leibo, Vinicius Zambaldi, Charles Beattie, Karl Tuyls, and Thore Graepel. A multi-agent reinforcement learning model of common-pool resource appropriation. *Advances in neural information processing systems*, 30, 2017.

[12] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

[13] Taiwei Shi, Sihao Chen, Bowen Jiang, Linxin Song, Longqi Yang, and Jieyu Zhao. Experiential reinforcement learning. *arXiv preprint arXiv:2602.13949*, 2026.

[14] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik R Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=vAElhFcKW6.

[15] Joar Max Viktor Skalse, Nikolaus H. R. Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward gaming. In Alice H. Oh, Alekh Agarwal, Danielle Belgrave, and Kyunghyun Cho, editors, *Advances in Neural Information Processing Systems*, 2022. URL https://openreview.net/forum?id=yb3HOXO3lX2.

[16] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=ehfRiF0R3a.

[17] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=Bb4VGOWELI.

[18] Haoran Ye, Jiarui Wang, Zhiguang Cao, Federico Berto, Chuanbo Hua, Haeyeon Kim, Jinkyoo Park, and Guojie Song. Reevo: Large language models as hyper-heuristics with reflective evolution. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=483IPG0HWL.

Appendix A. Environment Details

Gathering [6]. Agents navigate a 2D gridworld and collect apples ($+1$ reward). Apples respawn on a fixed 25-step timer. Agents may fire a tagging beam (2 hits remove a rival for 25 steps). The dilemma: agents can coexist peacefully and share resources, or attack rivals to monopolize apples, but aggression wastes time and reduces total welfare.

Cleanup [5]. A public goods game with two regions: a river that accumulates waste, and an orchard where apples grow. Apples only regrow when the river is sufficiently clean. Agents can fire a cleaning beam (costs -1) to remove waste, or collect apples ($+1$). A penalty beam (costs -1, inflicts -50 on the target) can tag rivals out for 25 steps. The dilemma: cleaning is costly but benefits everyone; purely selfish agents free-ride on others' cleaning.

Both games use 8–9 discrete actions (4 movement directions, 2 rotations, beam, stand, and optionally clean) and episodes of $H = 1000$ steps. Gathering uses a 38×16 gridworld with $\sim$120 apple spawns; Cleanup uses a gridworld with separate river and orchard regions.

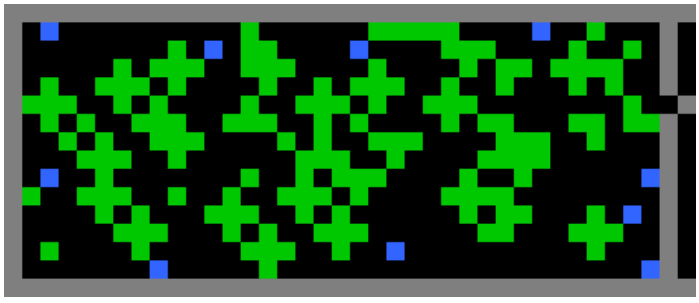


Figure 2: *
(a) Gathering

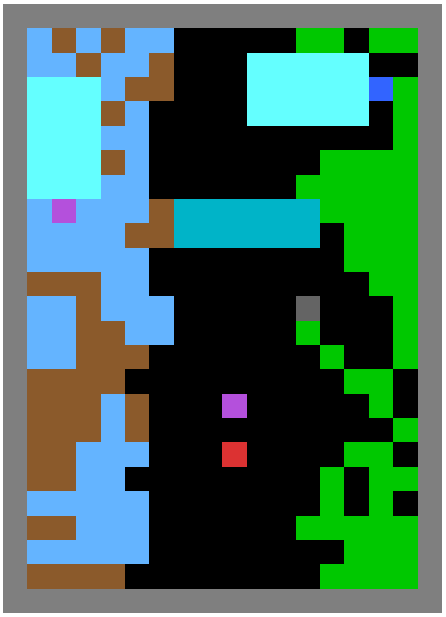


Figure 3: *
(b) Cleanup

Figure 4: Screenshots of the two SSD environments used in our experiments. **(a)** *Gathering*: agents (colored markers) navigate a gridworld to collect apples (green cells); apples respawn on a fixed timer and agents may fire tagging beams to temporarily remove rivals. **(b)** *Cleanup*: agents operate in two regions: a river accumulating waste (brown) and an orchard where apples grow (green); agents must cooperatively clean the river for apples to regrow. Agents may fire beams (cyan) to tag rivals out.

Appendix B. Baselines and Experimental Setup

Q-learner. Tabular Q-learning with a shared Q-table and non-trivial feature engineering: 7 hand-crafted discrete features in Gathering (BFS direction and distance to nearest apple, local apple density, nearest-agent direction and distance, beam-path check, own hit count; 4 320 states) and 8 features in Cleanup (adding BFS to nearest waste, global waste density, and a can-clean check; 11 664 states), plus cooperative reward shaping ($0.5 \cdot r_i + 0.5 \cdot \bar{r}$ − beam penalty, with an additional cleaning bonus in Cleanup). Trained for 1000 episodes with ε-greedy exploration.

BFS Collector. Hand-coded heuristic: BFS to the nearest apple, never beams or cleans.

GEPA. Genetic-Pareto Prompt Optimization [1], an LLM-based meta-optimizer that iteratively refines the *system prompt* (not the policy code) using a reflection LM. We run GEPA with Gemini 3.1 Pro for both generation and reflection, with $K = 3$ reflection iterations and $n_{\text{eval}} = 5$ evaluation seeds per candidate, matching the compute budget of the iterative code-level methods. GEPA's reflection LM receives only scalar reward; social metric definitions are not included, to avoid information leakage.

Appendix C. Generated Policy Analysis

All code excerpts below are verbatim LLM output, extracted from the best-performing iteration of representative runs. Comments in `[brackets]` are ours.

C.1. Gathering: Territory Strategies

Under dense feedback, the LLM discovers *BFS-Voronoi territory partitioning*: a multi-source flood-fill simultaneously from all alive agents computes true shortest-path ownership of every cell, correctly handling walls where Manhattan distance fails. The policy is purely cooperative—no agent ever fires the tagging beam.

Listing 1: Gathering — dense feedback (BFS-Voronoi, zero aggression).

```
# [Multi-source BFS: O(H*W) flood-fill assigns territory]
bfs_q = deque()
dist_map = {}

for i in range(env.n_agents):
    if int(env.agent_timeout[i]) > 0:
        continue
    r, c = int(env.agent_pos[i][0]), int(env.agent_pos[i][1])
    dist_map[(r, c)] = (0, i)
    bfs_q.append((r, c, 0, i))

while bfs_q:
    r, c, d, owner = bfs_q.popleft()
    cur = dist_map.get((r, c), (10**9, 10**9))
    if d > cur[0] or (d == cur[0] and owner > cur[1]):
        continue
    for dr2, dc2 in ((-1,0),(1,0),(0,-1),(0,1)):
        nr, nc = r + dr2, c + dc2
        if 0<=nr<H and 0<=nc<W and not walls[nr][nc]:
            nd = d + 1
            prev2 = dist_map.get((nr,nc), (10**9, 10**9))
            if nd<prev2[0] or (nd==prev2[0] and owner<prev2[1]):
                dist_map[(nr,nc)] = (nd, owner)
```

```
                bfs_q.append((nr, nc, nd, owner))

# Phase 1: BFS-navigate to nearest owned alive apple
# Phase 2: camp at nearest owned dead spawn (wait for respawn)
# Phase 3: global fallback to any reachable alive apple
```

Under sparse feedback, the LLM discovers *column-strip territory* (a simpler $O(1)$ assignment) but also develops a multi-tier combat system that wastes actions on beaming and chasing:

Listing 2: Gathering — sparse feedback (column strips + combat tiers).

```
# [Static column-strip territory: O(1) assignment]
zone_width = env.width / env.n_agents  # 38/10 = 3.8
zone_start = int(agent_id * zone_width)
zone_end = int((agent_id+1) * zone_width)

# [Multi-tier combat system wastes actions]
# Tier 1: Kill shot directly ahead -- always fire
# Tier 2: Rotate once to land kill shot
# Tier 3: Chase wounded opponents within range 8
# Tier 4: First hit on very close targets (range <= 2)

# [Evasion when one hit from being tagged]
if my_hits >= hits_to_tag - 1 and threats:
    # Flee away from nearest threat
    ...

# [Territory-based collection: home zone first, global fallback]
home_apples = {pos for pos in alive_apples
               if zone_start <= pos[1] < zone_end}
result = bfs_to_target_set(env, agent_id, home_apples)
```

The BFS-Voronoi policy achieves higher reward than column strips because territory adapts dynamically as agents move, and the absence of combat means every action is spent collecting.

C.2. Cleanup: Cleaner Allocation Strategies

The most impactful difference between dense and sparse policies is in *cleaner allocation*—how many agents are assigned to the costly but socially necessary cleaning role—and *cleaning efficiency*—how effectively each cleaner removes waste.

Under dense feedback, the LLM develops a waste-adaptive scaling schedule combined with optimized beam positioning:

Listing 3: Cleanup — dense feedback (adaptive scaling + optimal beam positioning).

```
# [Waste-adaptive schedule: up to 7/10 agents clean]
if   waste_ratio >= 0.8: n_cleaners = 7
elif waste_ratio >= 0.6: n_cleaners = 5
elif waste_ratio >= 0.4: n_cleaners = 3
elif waste_ratio >= 0.2: n_cleaners = 2
elif waste_ratio >= 0.07: n_cleaners = 1
else:                     n_cleaners = 0

# [Stable ID-based assignment: lowest IDs are permanent cleaners]
is_cleaner = agent_id < n_cleaners

# [Smart cleaning: search 9x9 area around waste centroid
#  for (row, col, orientation) maximizing waste in beam path]
cr, cc = int(np.mean(wr)), int(np.mean(wc))
for dr in range(-4, 5):
```

```
    for dc in range(-4, 5):
        r2, c2 = cr + dr, cc + dc
        if not env.walls[r2, c2]:
            for o in range(4):
                cnt = beam_count_at(r2, c2, o)
                if cnt > best_count:
                    best_count = cnt
                    best_pos = (r2, c2, o)
```

Under sparse feedback, the LLM assigns fixed cleaning roles to specific agents based on hard-coded thresholds, with simpler beam targeting:

Listing 4: Cleanup — sparse feedback (fixed agent-specific thresholds).

```
# [Only 4 of 10 agents can ever clean; hard-coded per-agent]
THRESHOLDS = {0: 0.15, 5: 0.20, 1: 0.40, 6: 0.45}
my_threshold = THRESHOLDS.get(agent_id, 2.0)  # 2.0 = never
is_cleaner = waste_fraction > my_threshold

# [Simple cleaning: best orientation from current position only]
counts = [count_waste_in_dir(o) for o in range(4)]
best_dir = counts.index(max(counts))
if best_dir == cur_orient:
    return CLEAN
```

The dense-feedback policy is more effective for two reasons: (1) it allocates *more cleaners when pollution is high*, preventing the ecosystem collapse that occurs when waste overwhelms a small fixed cleaning force; and (2) it *navigates to optimal firing positions* rather than cleaning only from the current location, dramatically increasing waste removal per action.

Appendix D. Prompts

This appendix reproduces the exact prompts used in all experiments. Each LLM call consists of a *system prompt* (fixed per game) and a *user prompt* (constructed per iteration). The system prompt defines the environment API, helper functions, and output format. The user prompt provides the iteration context: previous policy code and performance feedback.

The two feedback modes differ only in the user prompt at iterations $k \geq 1$:

- REWARD-ONLY (sparse): shows the previous policy source and scalar mean per-agent reward.
- REWARD+SOCIAL (dense): additionally includes social metric definitions and values (efficiency, equality, sustainability, peace).

At iteration 0, both modes use identical prompts since no history exists.

D.1. Gathering: System Prompt

The following system prompt is used for all Gathering experiments (both REWARD-ONLY and REWARD+SOCIAL).

Listing 5: Gathering system prompt (verbatim).

```
You are an expert game-theoretic AI designing policies for a multi-agent
Sequential Social Dilemma (the Gathering game).

## Environment Summary
```

```
- 2D gridworld. Agents collect apples (+1 reward each). Apples respawn after
  25 steps. Agents can fire a "tagging beam" that temporarily removes rivals
  for 25 steps (requires 2 hits to tag in Gathering).
- Episode length: 1000 steps.
- 8 actions: FORWARD(0), BACKWARD(1), STEP_LEFT(2), STEP_RIGHT(3),
  ROTATE_LEFT(4), ROTATE_RIGHT(5), BEAM(6), STAND(7)
- Agents move in 4 cardinal directions WITHOUT needing to rotate first
  (strafe movement). Rotation only matters for the beam direction.

## Environment API (available in your policy's namespace)

```python
# env attributes you can read:
env.agent_pos        # np.array shape (n_agents, 2) -- [row, col] per agent
env.agent_orient     # np.array shape (n_agents,) -- 0=N, 1=E, 2=S, 3=W
env.agent_timeout    # np.array shape (n_agents,) -- >0 means agent is removed
env.agent_beam_hits  # np.array shape (n_agents,) -- hits accumulated toward tag
env.apple_alive      # np.array shape (n_apples,) bool -- which apples exist
env._apple_pos       # np.array shape (n_apples, 2) -- [row, col] per apple spawn
env.walls            # np.array shape (H, W) bool -- wall map
env.height, env.width                  # map dimensions
env.n_agents, env.n_apples             # counts
env.beam_length, env.beam_width        # beam parameters (20, 1)
env.hits_to_tag, env.timeout_steps     # 2 hits to tag, 25 step timeout
```

Helper functions available in your namespace

```python
from gathering_env import Action, Orientation, _ROTATIONS, NUM_ACTIONS

bfs_nearest_apple(env, agent_id) -> Optional[Tuple[int,int]]
bfs_to_target_set(env, agent_id, target_set) -> Optional[Tuple[int,int]]
bfs_toward(env, agent_id, target_r, target_c) -> Optional[Tuple[int,int]]
direction_to_action(dr, dc, orientation) -> int
get_opponents(env, agent_id) -> list
_beam_targets_for_orient(env, ar, ac, orient_val, opponents) -> list
_rotation_distance(cur, target) -> int
greedy_action(env, agent_id) -> int
exploitative_action(env, agent_id) -> int
# Also available: np (numpy), deque (from collections)
```

Your task

Write a Python function called `policy` with this exact signature:

```python
def policy(env, agent_id) -> int:
    """Return an action (int 0-7) for the given agent."""
    ...
```

The function must:
1. Return an integer 0-7 (an Action value)
2. Be deterministic given the environment state
3. Only use the env attributes and helper functions listed above
4. Not import any modules (numpy and deque are pre-loaded)
5. Not use eval(), exec(), open(), or __import__

Working Example (seed BFS policy)

```python
def policy(env, agent_id) -> int:
    """BFS greedy: go to nearest apple, never beam."""
    if int(env.agent_timeout[agent_id]) > 0:
        return 7  # STAND while removed
    result = bfs_nearest_apple(env, agent_id)
    if result is None:
        return 7  # No reachable apple -- stand
    dr, dc = result
    return direction_to_action(dr, dc, int(env.agent_orient[agent_id]))
```

IMPORTANT:
- Always check `if result is None` before unpacking BFS results.
- Always cast env arrays to int when comparing.
- Always return a plain int (0-7), never a tuple or None.
- Put your code in a single ```python ... ``` block.
- Before the code block, explain your reasoning for the policy design.
```

D.2. Cleanup: System Prompt

The following system prompt is used for all Cleanup experiments.

Listing 6: Cleanup system prompt (verbatim).

````
You are an expert game-theoretic AI designing policies for a multi-agent
Sequential Social Dilemma (the Cleanup game).

## Environment Summary

- 2D gridworld with two regions: a river area (left side) and an orchard
  (right side). A stream separates the two regions.
- Agents collect apples in the orchard (+1 reward each).
- Waste (pollution) accumulates in the river over time.
- Episode length: 1000 steps.
- 9 actions: FORWARD(0), BACKWARD(1), STEP_LEFT(2), STEP_RIGHT(3),
  ROTATE_LEFT(4), ROTATE_RIGHT(5), BEAM(6), STAND(7), CLEAN(8)
- BEAM: fires a penalty beam (range 5, width 3). Costs -1 reward to fire.
  Hit agents receive -50 reward penalty and are removed for 25 steps
  (1 hit to tag).
- CLEAN: fires a cleaning beam (range 5, width 3). Costs -1 reward to fire.
  Removes waste cells in the beam's path, restoring clean river.
- Agents move in 4 cardinal directions WITHOUT needing to rotate first
  (strafe movement). Rotation only matters for the beam/clean direction.

## Environment API (available in your policy's namespace)

```python
# env attributes you can read:
env.agent_pos        # np.array shape (n_agents, 2) -- [row, col] per agent
env.agent_orient     # np.array shape (n_agents,) -- 0=N, 1=E, 2=S, 3=W
env.agent_timeout    # np.array shape (n_agents,) -- >0 means agent is removed
env.agent_beam_hits  # np.array shape (n_agents,) -- hits accumulated toward tag
env.apple_alive      # np.array shape (n_apples,) bool -- which apples exist
env._apple_pos       # np.array shape (n_apples, 2) -- [row, col] per apple spawn
env.walls            # np.array shape (H, W) bool -- wall map
env.waste            # np.array shape (H, W) bool -- True where waste exists
env.river_cells_set  # set of (row, col) -- all river cell positions
env.stream_cells_set # set of (row, col) -- stream cell positions
env.height, env.width                  # map dimensions
env.n_agents, env.n_apples             # counts
env.beam_length, env.beam_width        # beam/clean parameters (5, 3)
env.hits_to_tag, env.timeout_steps     # 1 hit to tag, 25 step timeout
```

## Helper functions available in your namespace

```python
from cleanup_env import CleanupAction, NUM_CLEANUP_ACTIONS
from gathering_env import Orientation, _ROTATIONS

bfs_nearest_apple(env, agent_id) -> Optional[Tuple[int,int]]
bfs_to_target_set(env, agent_id, target_set) -> Optional[Tuple[int,int]]
bfs_toward(env, agent_id, target_r, target_c) -> Optional[Tuple[int,int]]
direction_to_action(dr, dc, orientation) -> int
get_opponents(env, agent_id) -> list
_beam_targets_for_orient(env, ar, ac, orient_val, opponents) -> list
_rotation_distance(cur, target) -> int
greedy_action(env, agent_id) -> int
# Also available: np (numpy), deque (from collections)
```

## Your task

Write a Python function called `policy` with this exact signature:

```python
def policy(env, agent_id) -> int:
    """Return an action (int 0-8) for the given agent."""
    ...
```

The function must:
1. Return an integer 0-8 (a CleanupAction value)
2. Be deterministic given the environment state
3. Only use the env attributes and helper functions listed above
4. Not import any modules (numpy and deque are pre-loaded)
5. Not use eval(), exec(), open(), or __import__

## Working Example (seed BFS policy)

```python
def policy(env, agent_id) -> int:
    """BFS greedy: go to nearest apple, never beam or clean."""
````
````

```
    if int(env.agent_timeout[agent_id]) > 0:
        return 7  # STAND while removed
    result = bfs_nearest_apple(env, agent_id)
    if result is None:
        return 7  # No reachable apple -- stand
    dr, dc = result
    return direction_to_action(dr, dc, int(env.agent_orient[agent_id]))
```

IMPORTANT:
- Always check `if result is None` before unpacking BFS results.
- Always cast env arrays to int when comparing.
- Always return a plain int (0-8), never a tuple or None.
- Put your code in a single ```python ... ``` block.
- Before the code block, explain your reasoning for the policy design.

D.3. User Prompts

At each iteration k, a user prompt is constructed from the current state. Below we show the two templates.

D.3.1. ITERATION 0 (BOTH MODES)

When no prior policy exists, the user prompt is identical for both REWARD-ONLY and REWARD+SOCIAL. This is the ZERO-SHOT variant from the experiments:

Listing 7: User prompt at iteration 0 (both modes).

```
## Iteration 0/K: Write the initial policy

No prior policy exists yet. All agents will run the same code.
Your task is to write a first policy that maximizes per-agent reward.

## Instructions

Write a policy that maximizes per-agent reward. All agents will run your
exact same code simultaneously. There are {N} agents on a {W}x{H} map
with ~{A} apple spawns.
{env_hint}

Write your `policy(env, agent_id) -> int` function (returns 0-{max_action}).
```

where `{env_hint}` is game-specific:

- **Gathering**: *"Apples respawn every 25 steps. It takes 2 beam hits to tag out an agent."*
- **Cleanup**: *"Waste accumulates in the river over time. BEAM costs -1 to fire (-50 to target, 1 hit tags out for 25 steps). CLEAN costs -1 to fire (removes waste in beam path)."*

D.3.2. ITERATION $k \geq 1$: REWARD-ONLY (SPARSE)

The previous policy's source code and scalar reward are shown:

Listing 8: User prompt at iteration $k \geq 1$, REWARD-ONLY mode.

```
## Iteration k/K: Write an improved policy

The following policy is currently used by all agents. All agents run the
same code. Your task is to write an improved version that maximizes
per-agent reward.

### Current policy: **P{k-1}_reward**

```python
{previous_policy_source_code}
```

Results from previous iterations
```

```
- Iteration 0: Avg agent reward=X.X
- Iteration 1: Avg agent reward=Y.Y
...

## Instructions

Write a policy that maximizes per-agent reward. All agents will run your
exact same code simultaneously. There are {N} agents on a {W}x{H} map
with ~{A} apple spawns.
{env_hint}

Write your `policy(env, agent_id) -> int` function (returns 0-{max_action}).
```

D.3.3. ITERATION $k \geq 1$: REWARD+SOCIAL (DENSE)

In addition to the scalar reward, social metric definitions and values are included:

Listing 9: User prompt at iteration $k \geq 1$, REWARD+SOCIAL mode.

```
## Iteration k/K: Write an improved policy

The following policy is currently used by all agents. All agents run the
same code. Your task is to write an improved version that maximizes
per-agent reward.

### Current policy: **P{k-1}_rall**

```python
{previous_policy_source_code}
```

Results from previous iterations

Social Metrics (definitions)

- **Efficiency**: collective apple collection rate across all agents
  (higher = more apples collected per step).
- **Equality**: fairness of reward distribution between agents
  (1.0 = perfectly equal, negative = highly unequal).
- **Sustainability**: long-term apple availability -- measures whether
  resources are preserved over the episode (higher = apples remain
  available later in the episode).
- **Peace**: absence of aggressive beaming -- counts agents not involved
  in attack beam conflicts (higher = less aggression). Using the CLEAN
  beam to remove waste does NOT reduce peace.

- Iteration 0: Avg agent reward=X.X | efficiency=A.AAA,
  equality=B.BBB, sustainability=C.C, peace=D.D
- Iteration 1: Avg agent reward=Y.Y | efficiency=...
...

Instructions

Write a policy that maximizes per-agent reward. All agents will run your
exact same code simultaneously. There are {N} agents on a {W}x{H} map
with ~{A} apple spawns.
{env_hint}

Write your `policy(env, agent_id) -> int` function (returns 0-{max_action}).
```

Appendix E. Related Work (extended)

Sequential Social Dilemmas. Leibo et al. [6] introduced SSDs as temporally extended Markov games exhibiting cooperation–defection tension, instantiated in the Gathering gridworld. Hughes et al. [5] proposed the Cleanup game as a public goods variant requiring costly pro-social labor. Perolat et al. [11] formalized social outcome metrics (efficiency, equality, sustainability, peace) for evaluating multi-agent cooperation in SSDs.

LLMs for policy and program synthesis. FunSearch [12] uses LLMs to iteratively evolve programs that solve combinatorial optimization problems. Eureka [8] applies LLM code generation to design reward functions for robot control. Voyager [16] generates executable skill code for embodied

agents. Code as Policies [7] generates executable robot policy code from natural language via few-shot prompting; we extend this to multi-agent settings with iterative performance-driven refinement rather than one-shot instruction following. ReEvo [18] evolves heuristic algorithms through LLM reflection. Our work differs in applying LLM program synthesis to *multi-agent* environments, where policies must coordinate across agents sharing the same code.

LLM reflection and feedback. Reflexion [14] and Self-Refine [9] demonstrate that LLMs can self-improve through verbal feedback loops. OPRO [17] frames optimization as iterative prompt refinement. Shi et al. [13] introduce Experiential Reinforcement Learning (ERL), a training paradigm that embeds an experience–reflection–consolidation loop into reinforcement learning, where the model reflects on failed attempts and internalizes corrections via self-distillation. GEPA [1] combines reflective natural-language feedback with Pareto-based evolutionary search to optimize prompts, demonstrating that structured reflection on execution traces can outperform RL with substantially fewer rollouts. Our work specifically investigates how the *content* of evaluation feedback (scalar reward vs. multi-objective social metrics) affects the quality of LLM-generated multi-agent policies: what we call *feedback engineering*.

Reward hacking. When optimizing agents exploit unintended shortcuts in the reward signal or environment implementation, the resulting *reward hacking* [15] can produce high-scoring but undesirable behavior. Pan et al. [10] demonstrate that even small misspecifications in reward functions lead to qualitatively wrong policies. Goodhart's Law [4] ("when a measure becomes a target, it ceases to be a good measure") formalizes this risk. Our adversarial analysis (Section F) identifies a novel instantiation: LLM-generated policies that exploit environment state, a vulnerability absent from standard RL pipelines where the agent–environment boundary is enforced at the API level. Gallego [3] address in-context reward hacking at the specification level. This is complementary to our Section F finding that LLM-generated policies can exploit mutable environment state: SSC corrects the objective, while our analysis identifies the need to secure the environment interface.

Appendix F. Reward Hacking via Environment Mutation

Our framework gives policies programmatic access to the environment object (Section 2.2). We prompted Claude Opus 4.6 to analyze the environment and generate reward-hacking policies: it produced five distinct attacks autonomously, without guidance beyond the initial request. This demonstrates that the same models used for cooperative policy synthesis can equally discover exploits when prompted adversarially.

Mechanism. Each policy is called as $a_i = \pi(e, i)$ where e is the live environment instance. Our AST validator blocks dangerous operations (`eval`, file I/O, network access), but it cannot distinguish attribute *reads* from *writes*: any NumPy array attribute (`agent_pos`, `apple_alive`, `waste`, `agent_timeout`) is silently mutable, and no integrity check runs between the policy call and `env.step()`. The LLM identified two attack classes: *state manipulation* (teleporting onto apples, disabling rivals by setting their timeout to ∞) and *dynamics bypass* (clearing waste or force-spawning apples every step). All five attacks pass AST validation and the smoke test—they are valid `policy(env, agent_id) -> int` functions indistinguishable from legitimate policies at the interface level.

Table 2: Reward hacking via environment mutation in Cleanup ($N = 10$, large map, 1000 steps). Agent 0 runs the attack; agents 1–9 play the victim policy. Each cell shows Agent 0's reward and amplification over baseline.

Class	Attack	vs BFS	vs Optimized
—	Baseline	17 (1.0×)	104 (1.0×)
I: State	Teleport	34 (2.0×)	1000 (9.6×)
	Disable rivals	103 (6.1×)	103 (1.0×)
II: Dynamics	Purge waste	765 (45×)	765 (7.4×)
	Spawn apples	999 (59×)	999 (9.6×)
	Combined	1000 (59×)	1000 (9.6×)

Results. Table 2 reports results on the same Cleanup configuration as Table 1. Dynamics bypass attacks are dramatically more powerful than state manipulation: against BFS victims, teleporting yields only $2\times$ amplification (the bottleneck is apple respawn, not pathfinding), while force-spawning apples reaches the per-step theoretical maximum ($59\times$). Interestingly, better victims can *amplify* certain attacks: against optimized agents that actively clean waste, teleporting jumps from $2\times$ to $9.6\times$ because the attacker free-rides on their cleaning. Conversely, disabling optimized victims collapses the ecosystem (removing the cleaners leaves the attacker alone in a polluted map).

The most concerning finding connects directly to Section 3: dynamics bypass attacks that benefit all agents (purge waste, spawn apples) actually *improve* measured social metrics. Against BFS victims, "spawn apples" achieves the highest efficiency ($U = 5.99$) and sustainability ($S = 500.5$) of any configuration, surpassing every LLM-synthesized policy in Table 1. This illustrates a Goodharting [4] risk: a metric-optimizing LLM could discover dynamics manipulation as a "legitimate" strategy, as it maximizes social metrics while fundamentally violating the game's intended mechanics.

Implications. Standard mitigations exist (read-only proxies, state hashing, process isolation) but they highlight a deeper tension: the expressiveness that enables BFS pathfinding and territory partitioning (Section 3) is the same access that enables exploitation. Designing policy interfaces that are expressive enough for sophisticated coordination yet resistant to reward hacking remains an open challenge, and any verification pipeline must assume adversarial capability at least equal to the synthesizer's.