# CODING AGENTS ARE ZERO-SHOT COORDINATORS

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

We focus on zero-shot coordination (ZSC), where independently developed agents must cooperate with unseen partners at test time (e.g., autonomous driving). Self-play, a paradigm in which agents learn by interacting with copies of themselves, leads them to adopt specialised conventions that fail to generalise to new partners. Existing ZSC methods rely on restrictive assumptions and perform poorly on benchmarks such as Overcooked V2 and Yokai, achieving poor inter-seed cross-play (IXP), which measures mutual compatibility across independent runs of the same algorithm. Recent demonstrations of large-scale LLM coordination, together with rapid advancements in their coding capabilities, motivate a different approach: using general-purpose LLMs to directly generate ZSC policies as code. We find that coding agents, LLMs equipped with an off-the-shelf coding harness and runtime environment for iterative policy refinement, achieve state-of-the-art IXP on both Overcooked V2 and Yokai. Their policies also remain compatible across model families and novel Overcooked V2 configurations. Beyond ZSC, we consider a setting in which agents may pre-agree on conventions before deployment but are prohibited from joint training, as in self-play. Subsequently, we find that coding agents generate policies that reliably follow the H-group set of conventions used by human players in Hanabi. Taken together, our results show that coding agents significantly improve over existing ZSC methods, offering a promising route toward robust, adaptable, and human-compatible coordination.

## 1 INTRODUCTION

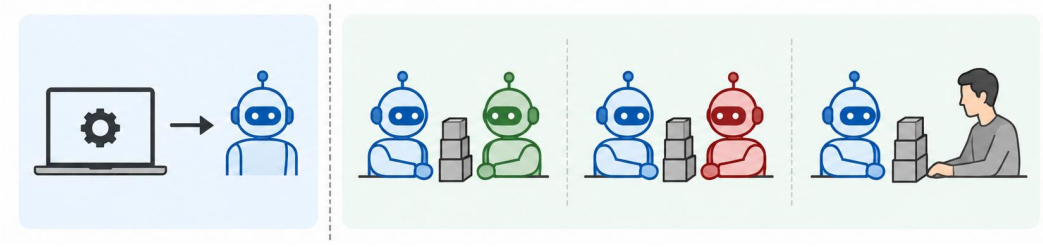


Figure 1: **Zero-shot coordination.** A policy is developed independently (left), then paired at test time with previously unseen agents or humans (right).

Building agents that can robustly cooperate with novel partners, particularly humans, is a central challenge in cooperative multi-agent reinforcement learning (MARL) (Dafoe et al., 2020; Hu et al., 2020), with important implications for real-world applications such as robotics (Nikolaidis et al., 2017; MacAlpine et al., 2014), autonomous driving (Sadigh et al., 2016; Cusumano-Towner et al., 2025), and communication (Foerster et al., 2016; Bard et al., 2020). This challenge is formalised by the zero-shot coordination (ZSC) setting, as illustrated in Figure 1. ZSC can be viewed as an equilibrium-selection problem: *when multiple high-payoff equilibria exist, how should an agent select a strategy that enables robust coordination with previously unseen partners?* Self-play (SP), a widely used MARL training paradigm (Samuel, 1959; Tesauro, 1994), does not resolve this problem. Since agents train exclusively with copies of themselves, they converge to highly specialised yet arbitrary conventions that achieve high payoffs in SP but fail to generalise to unseen partners.



Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Prior work (Hu et al., 2020; 2021; Cui et al., 2022; Forkel et al., 2026) has made significant progress on ZSC in complex environments such as Hanabi (Bard et al., 2020) and Overcooked (Carroll et al., 2019). However, many of these approaches are highly specialised and rely on restrictive assumptions or computationally expensive procedures, limiting their general applicability. These limitations are evident in benchmarks such as Overcooked V2 (Gessler et al., 2025) and Yokai (Ruhdorfer et al., 2026), where such methods are either not easily applicable or achieve low inter-seed cross-play (IXP), i.e., cross-play (XP) between policies trained with the same algorithm but different random seeds. Moreover, existing ZSC methods tend to *under-coordinate*: they adopt fixed strategies and do not support forming conventions with novel partners at test time (Gessler et al., 2025). While ZSC captures an important class of coordination problems, it does not encompass the full spectrum of real-world cooperation. Many settings lie between ZSC, which prohibits prior agreement on conventions, and self-play, which allows extensive joint training. In such intermediate settings, partners may agree on conventions before the start of the task without training together. For example, human Hanabi players often pre-agree on conventions (H-Group, n.d.) before the start of the game.

In this work, we investigate the use of large language models (LLMs) to address these challenges. Our motivation comes from their growing ability to tackle difficult scientific problems (OpenAI, 2026c; Bubeck et al., 2025) and coordinate in large multi-agent teams (OpenAI, 2026a). The recent breakthrough on the Navier–Stokes Millennium Prize Problem highlights a combination of these capabilities, with approximately 10,000 agents collaborating toward a solution (OpenAI, 2026b). In our setting, we use LLMs to generate policies as executable code, motivated by their rapidly advancing coding capabilities (Kwa et al., 2025) and recent success in synthesising control policies for robotics (Liang et al., 2023; Fu et al., 2026; Lu et al., 2026). Prior work (Hayler et al., 2026), using an older generation of models (e.g GPT 5.2) found that LLM-generated policies perform poorly at ZSC, even in simplified coordination tasks. We revisit this finding in light of significant advancements in the current frontier LLMs. Whereas prior work generated policies in a single forward pass, we use an off-the-shelf coding harness and runtime environment to enable iterative refinement through rollouts. We refer to these systems as coding agents.

**Our Contributions** We find that coding agents achieve state-of-the-art IXP returns on Yokai and Overcooked V2, surpassing the reported human baseline in Yokai. This compatibility extends across coding agents and to newly designed Overcooked V2 layouts, where policies generated by different LLMs continue to achieve strong XP. Coding agents also exhibit strong performance on Overcooked V2 tasks that require test-time convention formation, providing evidence that generated policies can mutually co-adapt at deployment. In Hanabi, generated policies achieve the highest reported SP returns among LLM-based approaches and competitive XP with human-proxy bots. When instructed to generate policies that follow the H-group set of conventions used by human players, coding agents achieve the highest XP returns among the evaluated methods with the Wilflame bot, a popular Hanabi community bot that strictly follows these conventions. Together, our findings establish coding agents as a promising approach for generating cooperative policies that generalise to unseen partners, adapt at test time, and are compatible with humans.

## 2 Related Work

**Learning to coordinate with unseen partners.** SP can produce strong cooperative policies (Foerster et al., 2019; Hu & Foerster, 2020), but independent training runs may converge to incompatible conventions. Several methods aim to address this by introducing structure into the learning process. Other-Play (Hu et al., 2020) uses environment symmetries to discourage arbitrary conventions, while entropy-regularised policy gradients (Forkel et al., 2026) can yield symmetry-equivariant policies under sufficiently strong regularisation. Off-Belief Learning (Hu et al., 2021) instead constrains how agents interpret past actions through a specified reference policy, enabling compatible reasoning across independent runs. Population-based methods (Strouse et al., 2021; Lupu et al., 2021; Sarkar et al., 2023; Lauffer et al., 2025) use diverse training populations to improve coordination with unseen partners. However, they rely on restrictive assumptions that prohibit general applicability. Other-Play requires prior specification of the environment's relevant symmetries, limiting its applicability when these symmetries are unknown or difficult to identify (Hu et al., 2020). OBL is also not model-free: its counterfactual updates require belief inference and simulator access to generate fictitious rollouts from sampled global states, which introduces significant computational

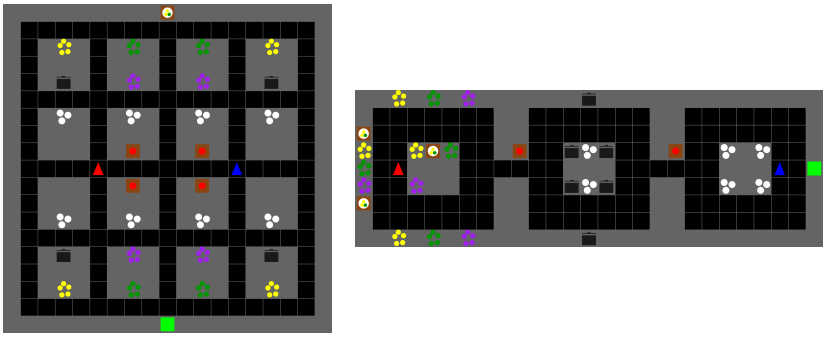


Figure 2: **Novel Overcooked V2 layouts.** We also evaluate coding agents on (a) *Block Grid* and (b) *Pipeline Cook*, two novel Overcooked V2 layouts not published in Gessler et al. (2025).

overhead (Hu et al., 2021). Population-based methods require large partner populations to capture diverse strategies, making training computationally demanding (Strouse et al., 2021). Forkel et al. (2026) show that, in certain environments, entropy regularisation sufficient to ensure symmetry equivariance yields poor returns even relative to other symmetry-equivariant policies. Recent benchmarks of Overcooked V2 (Gessler et al., 2025) and Yokai (Ruhdorfer et al., 2026) highlight these challenges where current methods exhibit significant gaps between SP and IXP returns. Certain Overcooked V2 layouts require establishing conventions online during interaction, a capability not supported by current ZSC methods (Gessler et al., 2025). Beyond IXP, a broader goal is coordination with human partners, sometimes evaluated directly in prior work (Hu et al., 2020). To reduce the cost and improve the reproducibility of such evaluations, Dizdarević et al. (2025) introduce AH2AC2, using human-proxy agents trained on over 147,000 human-played Hanabi games.

**Following shared conventions.** In certain coordination settings, partners may agree to follow a fixed set of conventions without having trained together; this includes human Hanabi players pre-agreeing to play according to the H-group set of conventions (H-Group, n.d.). The challenge is then to produce policies compatible with unseen partners, including humans that adopt the same conventions. InstructRL (Hu & Sadigh, 2023) uses language-model priors to regularise deep reinforcement learning (RL) toward simple hint conventions in Hanabi. Bredell et al. (2025) incorporate H-group-derived conventions into an augmented action space of a policy trained using deep RL. AH2AC2 (Dizdarević et al., 2025) instead prompts an LLM to select actions directly from game observations and descriptions of H-group Level 1 conventions. InstructRL and AH2AC2 both incur LLM inference costs for action-level guidance, while the approach of Bredell et al. (2025) requires manually implementing convention logic. In contrast, coding agents translate convention descriptions directly into executable policies in this work.

**LLMs for policy generation and optimisation.** Prior work has explored LLM-based cooperation using specialised agent scaffolds in Overcooked and Hanabi (Zhang et al., 2024a; Agashe et al., 2025; Ramesh et al., 2026). Hayler et al. (2026) study generic setups in which LLMs either select actions directly or generate executable policies, reporting coordination failures even in simple environments, particularly for code-generated policies. Their code-policy setup relies on single-step generation, whereas we use an off-the-shelf coding harness and runtime environment that support iterative policy generation and refinement through environment rollouts. This connects our approach to iterative refinement and LLM-based optimisation, where candidate solutions are revised using feedback (Madaan et al., 2023; Yang et al., 2024; Novikov et al., 2025). Whereas methods such as OPRO and AlphaEvolve evaluate candidates directly against a well-specified objective, the coding agents in our setting optimise an underspecified coordination objective without direct feedback on cross-play performance during policy generation and refinement.

## 3 Background

We focus on fully cooperative multi-agent settings that can be modelled as a decentralised partially observable Markov decision process (Dec-POMDP) (Bernstein et al., 2002). A Dec-POMDP $\mathcal{M}$ is defined as the tuple $(\mathcal{S}, n, \{\mathcal{A}^i\}_{i=1}^n, \{\mathcal{O}^i\}_{i=1}^n, \mathcal{T}, \mathcal{R}, T, \gamma, \rho_0)$, where $\mathcal{S}$ denotes the state space, $n$

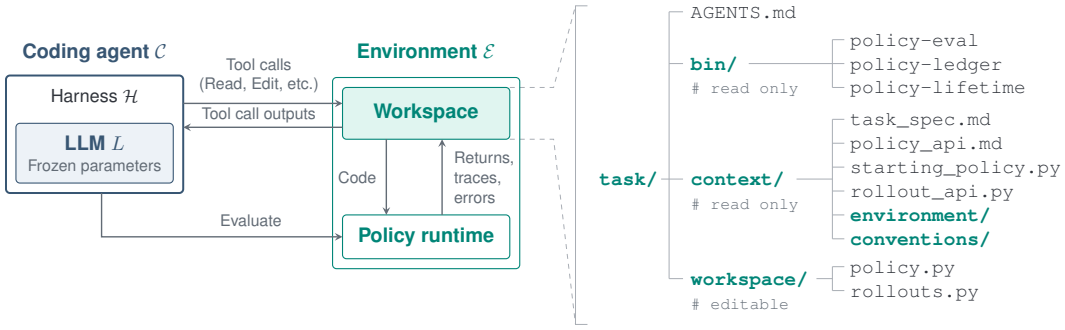



Figure 3: **Policy generation setup.** (Left) A frozen LLM uses harness-provided tools to construct, evaluate, and revise an executable joint policy. (Right) Initial environment state, comprising read-only task materials and tools alongside editable policy and rollout code. Note that the conventions directory is provided only in the Hanabi H-group condition.

the number of agents, $\mathcal{A}^i$ and $\mathcal{O}^i$ the action and observation spaces of agent $i$, respectively, $\mathcal{T}$ the transition function, $\mathcal{R}$ the shared reward function, $T$ the time horizon, $\gamma$ the discount factor, and $\rho_0$ the initial-state distribution. The joint action space is given by $\mathcal{A} = \prod_{i=1}^{n} \mathcal{A}^i$. At timestep $t$, the environment is in state $s_t \in \mathcal{S}$ and the agents jointly select an action $\mathbf{a}_t = (a_t^1, \ldots, a_t^n) \in \mathcal{A}$, after which the environment transitions according to $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \Delta(\mathcal{S})$ and the agents receive a shared reward given by $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. Each agent $i$ receives only a local observation $o_t^i = \mathcal{O}^i(s_t)$ and acts based on its local action-observation history (AOH) $\tau_t^i = (o_0^i, a_0^i, \ldots, a_{t-1}^i, o_t^i)$. A local policy $\pi^i(a_t^i \mid \tau_t^i)$ specifies the probability with which agent $i$ selects action $a_t^i$ given its AOH. We denote the joint policy by $\boldsymbol{\pi} = (\pi^1, \ldots, \pi^n)$, such that the joint action distribution factors as $\boldsymbol{\pi}(\mathbf{a}_t \mid \boldsymbol{\tau}_t) = \prod_{i=1}^{n} \pi^i(a_t^i \mid \tau_t^i)$. A learning algorithm $\mathcal{L}$ maps a Dec-POMDP $\mathcal{M}$ to a distribution over joint policies. The SP return of a joint policy $\boldsymbol{\pi}$ is defined as the expected discounted return obtained when each agent $i$ acts according to its corresponding local policy $\pi^i(\cdot \mid \tau_t^i)$:

$$J_{\text{SP}}(\boldsymbol{\pi}) = \mathbb{E}_{\boldsymbol{\pi}} \left[ \sum_{t=0}^{T-1} \gamma^t \mathcal{R}(s_t, \mathbf{a}_t) \right].$$

Given a Dec-POMDP $\mathcal{M}$ with $n$ agents, let $\boldsymbol{\pi}_1, \ldots, \boldsymbol{\pi}_n$ be joint policies sampled as $\boldsymbol{\pi}_j = (\pi_j^1, \ldots, \pi_j^n) \sim \mathcal{L}_j(\mathcal{M})$. The XP return is defined as:

$$J_{\text{XP}}(\boldsymbol{\pi}_1, \ldots, \boldsymbol{\pi}_n) := \mathbb{E}_{\phi \sim \text{Perm}(n)} \left[ J_{\text{SP}} \left( \pi_{\phi(1)}^1, \ldots, \pi_{\phi(n)}^n \right) \right],$$

where $\text{Perm}(n)$ is the set of permutations of $\{1, \ldots, n\}$. We define *inter-seed cross-play* (IXP) as XP between policies produced by independent runs of the same learning algorithm with different random seeds. High IXP is a necessary, but not sufficient, condition for ZSC: if an algorithm does not produce mutually compatible policies across its own training runs, we cannot expect those policies to coordinate reliably with unseen partners. Despite this limitation, IXP is widely used to compare learning algorithms for ZSC (Hu et al., 2020; 2021; Forkel et al., 2026).

## 4 Policy Generation Setup

We do not propose coding agents as a novel learning algorithm; instead, we evaluate whether they can generate effective policies for zero-shot coordination. We define a coding agent as a pair $\mathcal{C} = (L, \mathcal{H})$, consisting of an LLM $L$ and a harness $\mathcal{H}$. The harness connects the model to a development environment $\mathcal{E}$ by providing tools for interacting with it, including file editing and code execution. The model interacts with the environment through successive tool calls, conditioned on the task instructions and interaction history. The harness executes these calls and returns their outputs to the model. We use Codex and Claude Code harnesses for OpenAI and Anthropic models, respectively.

In our setting, $\mathcal{E}$ contains a sandboxed workspace for storing policy code and experimental artefacts, together with a runtime for executing and evaluating candidate policies. The runtime provides access to a GPU to accelerate policy evaluation. The workspace provides a task description specifying the

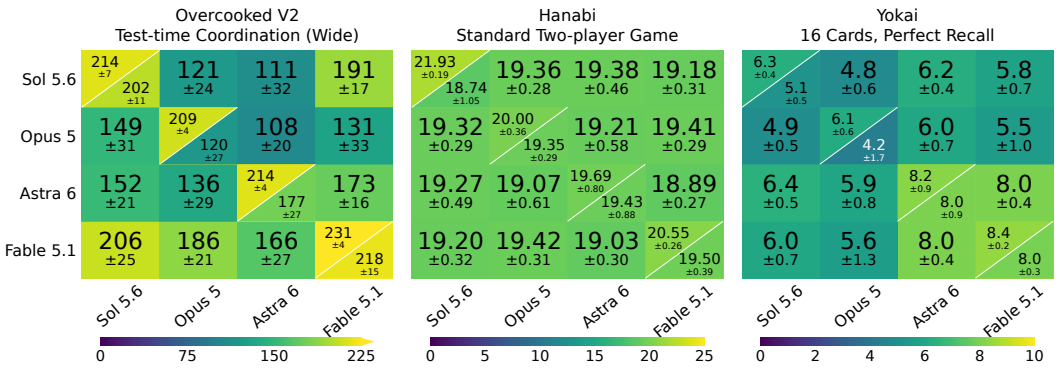



Figure 4: **Inter-model cross-play with SP and IXP diagonals.** Diagonal cells show SP in the upper-left triangle and IXP in the lower-right triangle. Each model contributes six policies: SP averages six same-policy pairings, while XP and IXP average all 30 ordered unequal-seed pairings, each evaluated over 1,024 games. Cells report mean ± SEM.

game, the learning objective, and the policy API. The game description includes the rules, configuration, each player's observations and available actions, the shared reward, and episode termination conditions. The objective is to construct policies that coordinate robustly with independently generated partners. The workspace also includes a default policy template implementing the policy and rollout APIs for evaluating candidates. We do not provide any solution strategies, except for the conventions provided in the Hanabi H-group condition, and the agent has no access to the simulator source code or the internet through its tools. The policy API enforces decentralised execution: each player acts using only its local observation and private memory. The coding agent determines how this memory is represented and updated, allowing policies to retain information from past observations. To accommodate asymmetric player roles in Overcooked V2, the coding agent implements a separate local policy for each role. In Hanabi and Yokai, where the roles are symmetric, it implements a single local policy, with SP pairing two instances that maintain separate private memories.

With $L$ and $\mathcal{H}$ fixed, the coding agent induces a stochastic learning algorithm that takes a task description and a computational budget as inputs and returns an executable joint policy as code. It interacts with $\mathcal{E}$ to implement candidate policies and evaluate them through rollouts, using the resulting feedback to guide further revisions. This iterative experimentation constitutes the learning process, while the language model's parameters remain fixed throughout. To make this process interpretable, we provide an experiment ledger tool whose API is documented in the task description. We enforce that the agent use this tool to record the hypothesis behind each revision, its evaluation results, and the stated rationale for retaining or rejecting it.

Each independent run of the learning algorithm, which we sometimes refer to as an independent seed, is initialised in a fresh session and workspace with no access to the policies, interaction histories, or evaluation results of other seeds. Each seed receives a distinct paraphrase of the task specification and objective, with their meaning preserved, reflecting our assumption that independently generated coding agents need not receive identically worded prompts. We allocate a two-hour wall-clock budget to each policy-generation run and make the elapsed time available to the agent. After each trial, we instruct the agent to update its selection, retaining the candidate it judges most likely to maximise the XP objective. The agent may terminate early; its last selected policy constitutes the output of the learning algorithm. During evaluation, this policy is executed without further LLM calls. Figure 3 illustrates the learning process and initial workspace contents; Appendix A provides the objective and final policy selection instructions.

## 5 Experiments

We evaluate coding agents on Overcooked V2, Yokai, and Hanabi, which pose coordination challenges not addressed by current ZSC methods (Gessler et al., 2025; Ruhdorfer et al., 2026; Dizdarević et al., 2025). We use their corresponding public JaxMARL-based implementations (Rutherford et al., 2024). We use multiple ChatGPT Pro and Claude Code Max subscriptions for our experi-

Table 1: **Coordination performance on Overcooked V2.** Mean episode return, with ± SD shown below each mean. Coding-agent results use six seeds and 1,024 games per policy pairing; published baselines, taken directly from Gessler et al. (2025), use ten seeds and 500 games per pairing. We find that coding agents achieve the highest IXP return on all six evaluated layouts.

| | Demo Cook | | | | Grounded Coord. | | | | Test-time | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Simple | | Wide | | Simple | | Ring | | Simple | | Wide | |
| Algorithm $\mathcal{L}$ | SP | IXP | SP | IXP | SP | IXP | SP | IXP | SP | IXP | SP | IXP |
| Self-Play | 172 ± 19 | 36 ± 82 | 195 ± 4 | 40 ± 61 | 154 ± 10 | −55 ± 124 | 164 ± 10 | −12 ± 75 | 145 ± 22 | −81 ± 99 | 194 ± 12 | −30 ± 96 |
| State-Augmented | 170 ± 16 | 112 ± 50 | 192 ± 7 | 60 ± 60 | 156 ± 29 | −17 ± 115 | 165 ± 7 | −16 ± 65 | 161 ± 18 | −55 ± 103 | 137 ± 82 | −12 ± 72 |
| Other-Play | 149 ± 25 | 47 ± 48 | 162 ± 23 | 62 ± 53 | 122 ± 24 | 4 ± 73 | 121 ± 36 | −9 ± 21 | 121 ± 37 | −3 ± 51 | 175 ± 54 | −11 ± 42 |
| Fictitious Co-Play | 22 ± 48 | 1 ± 25 | 45 ± 43 | 13 ± 28 | 64 ± 58 | 46 ± 47 | 86 ± 34 | 6 ± 46 | 35 ± 44 | 6 ± 29 | 95 ± 56 | 23 ± 40 |
| GPT-5.6 Sol | 207 ± 16 | 174 ± 27 | 185 ± 32 | 162 ± 29 | 149 ± 18 | 129 ± 9 | 152 ± 17 | 133 ± 14 | 180 ± 1 | 155 ± 19 | 214 ± 17 | 202 ± 11 |
| Claude Opus 5 | 198 ± 4 | 197 ± 1 | 202 ± 4 | 193 ± 8 | 139 ± 8 | 130 ± 5 | 147 ± 4 | 130 ± 4 | 172 ± 6 | 168 ± 2 | 209 ± 11 | 120 ± 26 |
| GPT-6 Astra | 198 ± 3 | 191 ± 7 | 213 ± 9 | 211 ± 4 | 138 ± 4 | 137 ± 1 | 152 ± 3 | 150 ± 2 | 161 ± 12 | 155 ± 5 | 214 ± 11 | 177 ± 26 |
| Claude Fable 5.1 | 200 ± 11 | 196 ± 1 | 212 ± 16 | 203 ± 5 | 134 ± 3 | 134 ± 1 | 158 ± 16 | 130 ± 7 | 174 ± 9 | 172 ± 4 | 231 ± 10 | 218 ± 15 |

ments, costing approximately US$800 per month. Subscription accounts are significantly subsidised relative to standard API rates (SemiAnalysis, 2026), making large-scale experimentation feasible. We therefore restrict our study to OpenAI and Anthropic models, as comparable experiments with other frontier models at standard API rates were prohibitively expensive. The reasoning effort is always set to `high`. In the Hanabi H-group setting, we provide an offline adaptation of the H-group Level 1 documentation (H-Group, n.d.), including worked examples.

### 5.1 Self-play and inter-seed XP

As shown in Tables 1 and 2, coding agents obtain strong self-play scores in multiple configurations of the Overcooked V2 and Yokai environments. In certain Yokai configurations (9C imperfect recall and 16C perfect recall), existing methods even fail to achieve meaningful self-play scores, whereas coding agents remain effective. More importantly, coding agents achieve state-of-the-art inter-seed cross-play performance across every tested environment configuration in OvercookedV2 and Yokai, demonstrating that independently generated policies remain mutually compatible. Strong IXP on Overcooked V2's test-time coordination layouts provides evidence that the generated policies can mutually co-adapt to establish conventions at test-time (Table 1). In Yokai's nine-card imperfect-memory setting, Claude Fable 5.1 and GPT 6 Astra achieve IXP returns of 6.9 and 6.5, respectively, both exceeding the reported human baseline of 5.7 (Ruhdorfer et al., 2026). In Hanabi (Table 3), coding agents achieve self-play returns of up to 21.93, surpassing previously reported two-player results of 17.5 (Ramesh et al., 2026) and 13.33 (Agashe et al., 2025) by LLM based methods, both of which rely on specialised scaffolds.[1] While coding agents outperform the IPPO (SP) baseline under IXP, they remain below existing ZSC methods in both SP and IXP. We provide examples of generated policy code in Appendix G, including a Claude Fable 5.1 policy for Overcooked V2's Test-time (Wide) layout that supports test-time convention formation by updating beliefs over its partner's signalling conventions during play (Appendix G.4). We also provide experiment-ledger entries in Appendix H. Interestingly, these entries show Claude Fable 5.1 constructing alternative partner policies to test the cross-play robustness of its candidate policy. We provide learning curves (Appendix Figure 11) that show improvements in SP and IXP returns over the course of policy generation. We also evaluate GPT-5.6 Luna, Claude Sonnet 5, and GPT-5.5, with results reported in

[1]Both works retain accumulated scores when all lives are lost, whereas we assign zero; Agashe et al.'s best-performing configuration reports no bomb-outs.

Table 2: **Coordination performance on two-player Yokai.** Mean episode return ± SEM. Coding-agent results use six seeds and 1,024 games per policy pairing; the 12-seed method baselines are taken directly from Ruhdorfer et al. (2026), were evaluated on 5,000 games. They also report an illustrative human-human baseline of $5.7 \pm 0.5$ across 25 games in the nine-card imperfect-memory setting. We find that coding agents achieve the highest IXP in all three configurations; their best nine-card imperfect-memory IXP also exceeds this human reference.

| Configuration | Algorithm $\mathcal{L}$ | SP | IXP |
|---|---|---|---|
| 9 cards Perfect memory | IPPO | $7.5 \pm 0.3$ | $1.6 \pm 0.2$ |
| | IPPO+High Ent. | $7.5 \pm 0.1$ | $4.1 \pm 0.8$ |
| | Other-Play | $6.6 \pm 0.2$ | $4.8 \pm 0.1$ |
| | OBL L4 | $4.5 \pm 0.2$ | $2.8 \pm 0.3$ |
| | GPT-5.6 Sol | $4.9 \pm 0.3$ | $4.7 \pm 0.1$ |
| | Claude Opus 5 | $6.0 \pm 0.5$ | $5.6 \pm 0.3$ |
| | GPT-6 Astra | $7.5 \pm 0.5$ | $7.3 \pm 0.2$ |
| | Claude Fable 5.1 | $7.0 \pm 0.3$ | $7.0 \pm 0.1$ |
| 9 cards Imperfect memory | Other-Play | $0.0 \pm 0.0$ | $0.0 \pm 0.0$ |
| | GPT-5.6 Sol | $4.9 \pm 0.5$ | $4.3 \pm 0.1$ |
| | Claude Opus 5 | $5.2 \pm 0.5$ | $5.2 \pm 0.2$ |
| | GPT-6 Astra | $6.3 \pm 0.5$ | $6.5 \pm 0.2$ |
| | Claude Fable 5.1 | $7.1 \pm 0.3$ | $6.9 \pm 0.2$ |
| 16 cards Perfect memory | IPPO | $0.0 \pm 0.0$ | $0.0 \pm 0.0$ |
| | GPT-5.6 Sol | $6.3 \pm 0.4$ | $5.1 \pm 0.2$ |
| | Claude Opus 5 | $6.1 \pm 0.6$ | $4.2 \pm 0.7$ |
| | GPT-6 Astra | $8.2 \pm 0.9$ | $8.0 \pm 0.4$ |
| | Claude Fable 5.1 | $8.4 \pm 0.2$ | $8.0 \pm 0.1$ |

Tables 4, 6, and 7 in the Appendix. Weaker models achieve lower SP and IXP returns, indicating that state-of-the-art IXP observed in our experiments is currently limited to the most capable models.

### 5.2 XP WITH HETEROGENEOUS PARTNERS

We next evaluate whether compatibility extends beyond independent runs of the same coding agent. As shown in Figure 4 and Figure 10 in the Appendix, policies generated by different coding agents remain compatible. Across the tested Overcooked V2 and Yokai environment configurations, inter-model XP exceeds the best reported IXP baseline. Some pairings nevertheless show gaps between SP and inter-model XP, indicating that there still exists room for improvement. We also find that inter-model XP tends to be highest between the most capable models, with the GPT-6 Astra and Claude Fable 5.1 pairing achieving the highest scores across the evaluated environments.

In Hanabi, we evaluate cross-play with two AH2AC2 human-proxy bots (Dizdarević et al., 2025) and the Wilflame bot (will-hanabi-bot, n.d.), configured to follow H-group Level 1 conventions. Against the human proxies, Claude Fable 5.1 achieves a return of $19.34 \pm 0.68$, comparable to $19.70 \pm 0.16$ for OBL L4 and above IPPO, IPPO+HE, and OBL L6 (Table 3). Despite similar returns with the human proxies, Fable and OBL L4 differ sharply with Wilflame: without convention instructions, Fable scores 14.87, compared with 5.19 for OBL L4. Providing H-group Level 1 descriptions to Fable increases its XP return to 20.25, close to Wilflame's SP return of 21.05, indicating strong compatibility with its convention-based play.

### 5.3 EVALUATION ON NOVEL ENVIRONMENTS

The strong ZSC performance of coding agents raises questions of whether they benefit from prior exposure to the evaluated tasks, a concern also raised in other domains (Zhang et al., 2024b; Wu et al., 2024; Jain et al., 2025). Unlike the ZSC baselines trained from scratch, coding agents can draw on knowledge acquired during pretraining, which may include, for example, information about the Overcooked V2 layouts and their associated coordination challenges from the published literature. Hence, we evaluate coding agents on two novel configurations, the *Block Grid* and *Pipeline Cook* layouts (Figure 2). Both these layouts are more complex than the original benchmark layouts and, unlike the latter, do not readily admit effective ZSC strategies. Nevertheless, coding agents exhibit

Table 3: **Coordination performance on two-player Hanabi.** Mean ± SEM over six coding-agent runs and four checkpoints per baseline, with 1,024 games per policy pairing or policy-bot seat assignment. We evaluate R2D2-based OBL checkpoints from Hu et al. (2021) and IPPO, IPPO+HE, and IPPO-based OBL checkpoints from Forkel et al. (2026). We find that coding agents underperform the ZSC baselines in SP and IXP, but achieve significantly higher XP with the Wilflame bot.

| Algorithm $\mathcal{L}$ | Setting | SP | IXP | Human Proxy Bot XP (SP: 23.80) | Wilflame Bot XP (SP: 21.05) |
|---|---|---|---|---|---|
| IPPO | Standard | 24.53 ± 0.02 | 15.63 ± 2.13 | 13.60 ± 1.54 | 3.96 ± 0.48 |
| IPPO+HE | Standard | 24.48 ± 0.01 | 24.43 ± 0.02 | 11.64 ± 0.09 | 4.41 ± 0.08 |
| OBL L4 (R2D2) | Standard | 24.12 ± 0.03 | 23.76 ± 0.08 | 19.70 ± 0.16 | 5.19 ± 0.04 |
| OBL L6 (IPPO) | Standard | 24.35 ± 0.01 | 24.32 ± 0.00 | 14.92 ± 0.15 | 5.56 ± 0.13 |
| GPT-5.6 Sol | Standard | 21.93 ± 0.19 | 18.74 ± 0.42 | 15.53 ± 2.25 | 7.42 ± 1.15 |
| | H-group L1 | 18.59 ± 0.65 | 18.26 ± 0.29 | 16.92 ± 0.57 | 18.54 ± 0.47 |
| Claude Opus 5 | Standard | 20.00 ± 0.36 | 19.35 ± 0.12 | 18.60 ± 0.36 | 12.38 ± 0.63 |
| | H-group L1 | 20.05 ± 0.42 | 19.09 ± 0.20 | 17.75 ± 0.24 | 18.21 ± 0.63 |
| GPT-6 Astra | Standard | 19.69 ± 0.80 | 19.43 ± 0.35 | 17.02 ± 1.61 | 12.09 ± 1.55 |
| | H-group L1 | 21.53 ± 0.34 | 21.11 ± 0.19 | 17.87 ± 0.33 | 19.98 ± 0.25 |
| Claude Fable 5.1 | Standard | 20.55 ± 0.26 | 19.50 ± 0.16 | 19.34 ± 0.68 | 14.87 ± 0.99 |
| | H-group L1 | 21.81 ± 0.21 | 20.80 ± 0.12 | 18.17 ± 0.40 | 20.25 ± 0.28 |

strong performance on these layouts, generating policies that remain compatible across independent generations (Appendix Table 5) and model types (Figure 5). These results provide evidence of coding agents' genuine ability to generate robust policies, rather than relying on leaked knowledge of environments and their solution strategies. That said, disentangling the precise contribution of knowledge acquired during pretraining on ZSC performance is beyond the scope of this study.

### 5.4 Policy generation capabilities

We next examine how the general capabilities of the LLM affect the policy generation performance of the coding agent. We compare two policy-generation objectives: maximising SP return and maximising XP return with independently generated partners (as shown in Appendix Figures 9 and 6). This tests whether the lower returns of weaker models under the XP objective stem from limited policy-generation capability or specifically from difficulty generating ZSC compatible policies. In particular, we ask whether these models can still produce high-return joint policies when permitted to exploit specialised conventions. In this experiment, we retain the same model, harness and computational budget, changing only the objective and selection rule. Under the SP objective, we re-evaluate each proposed policy, selecting the one with the highest SP return; under the XP objective, the model selects the final policy. Figure 5 (left) compares the resulting policies across the six Overcooked V2 layouts. We find that more capable models achieve higher SP returns under the SP objective while exhibiting a larger SP–IXP gap, indicating a greater capacity to exploit specialised conventions. Under the XP objective, they likewise achieve higher IXP returns, reflecting improved compatibility across independent generations. At the same time, even less capable models can generate policies whose SP returns exceed the IXP returns achieved by more capable models. Together, these results suggest that increasing model capability broadens the range of effective policies accessible to coding agents, spanning both highly specialised and robust ZSC policies.

## 6 Limitations and future work

**Performance in Hanabi.** Coding agents remain below existing ZSC methods in Hanabi SP and IXP, despite their strong results on Overcooked V2 and Yokai. This gap may reflect difficulties in representing effective belief tracking and decision rules through code. To overcome this, future work could combine generated policy code with learned neural components, drawing on neuro-symbolic approaches (Gaunt et al., 2017; Anderson et al., 2020), or use LLM-derived priors to guide learning of neural network policies, similar to InstructRL (Hu & Sadigh, 2023).

**Evaluation beyond IXP.** As discussed in Section 3, IXP measures compatibility between independent runs of the same learning algorithm, but it does not fully capture broader coordination with

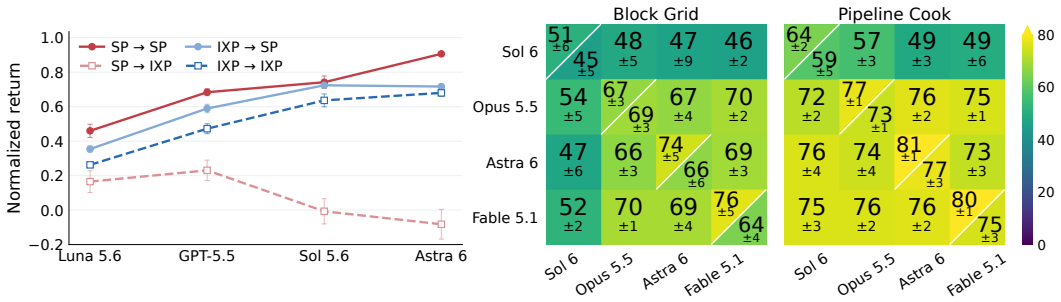



Figure 5: **(Left) Normalised SP and IXP returns for policies generated under SP or IXP objectives** (legend: objective → evaluation), averaged across the six default environments; error bars show ± SEM. Models are ordered from left to right by increasing Artificial Analysis Intelligence Index score (Artificial Analysis, n.d.). **(Middle, right) Inter-model XP on *Block Grid* and *Pipeline Cook*.** Diagonal cells show SP in the upper-left triangle and IXP in the lower-right triangle. Each model contributes six policies: SP averages six same-policy pairings, while XP and IXP average all 30 ordered unequal-seed pairings, each evaluated over 1,024 games. Cells report mean ± SEM.

unseen partners. As discussed in Section 5.2, despite weaker Hanabi IXP, coding agents achieve cross-play comparable to ZSC baselines with the human-proxy bot and significantly stronger cross-play with the Wilflame bot. This motivates further benchmarks similar to AH2AC2 beyond Hanabi, assessing compatibility with human proxies and other diverse partners not seen during training.

**Shared prior knowledge.** Coding agents do not learn from scratch: shared knowledge acquired during pretraining may facilitate IXP, inter-model XP, and XP with publicly available bots. Our novel-layout experiments reduce concerns about familiarity with the tested environment but do not isolate the contribution of shared priors. In Hanabi, coding agents' strong XP with Wilflame (Appendix Table 7) may reflect both partner robustness and prior knowledge of human conventions. We find some evidence for the latter, including references to H-group-related concepts such as "chop" and "save"; on one occasion, Fable explicitly states that *"a rank-5 hint that touches my oldest untouched card is a save in most conventions"*. Note that this is in the standard setting, where no H-group material is supplied. That said, we find no explicit mention of H-group in any of the final policies or experiment traces. These observations suggest that relevant knowledge of the partner's conventions may already be encoded in the models, making it difficult to strictly enforce zero-shot coordination evaluation when working with coding agents. Nevertheless, in the real world, humans who have never previously interacted regularly coordinate by drawing on shared background knowledge, including cultural conventions. Future work could examine how knowledge acquired during pre-training affects the performance of coding agents in ZSC.

## 7 Conclusion

In this work, we investigated coding agents as generators of executable policies for ZSC. We found that frontier models equipped with off-the-shelf coding harnesses and runtime environments achieve state-of-the-art IXP on Overcooked V2 and Yokai, with compatibility extending across model families and novel Overcooked V2 layouts. Their performance on tasks requiring test-time convention formation provides evidence that generated policies can adapt their coordination strategies during interaction. In Hanabi, coding agents achieve competitive cross-play with human-proxy bots, while supplying H-group Level 1 conventions enables strong cooperation with Wilflame without joint training. Nevertheless, their Hanabi SP and IXP remain below existing ZSC methods. We find that policy generation capability under both SP and XP objectives improves with model capability, suggesting that further increases in model capability may reduce the remaining performance gap in Hanabi and the coordination gap in Overcooked V2. We note that shared knowledge acquired during pretraining may also facilitate cross-play, much as unfamiliar humans coordinate through common background knowledge. Together, these findings establish coding agents as a promising approach to coordination-policy generation and motivate evaluation frameworks that distinguish the benefits of shared priors from the ability to adapt to unfamiliar partners.

### AI USE STATEMENT

In this work, we used generative AI tools to implement methods, namely Codex (w/ GPT 5.6 Sol, GPT 6 Sol, and GPT 6 Astra) to implement the coding agent sandbox for policy generation and to orchestrate, monitor and provide feedback on experiments. We have also used GPT 6 Astra to help interpret the policies generated by the coding agents. We have not used generative AI tools to propose or refine hypotheses. The use of generative AI to generate synthetic data sets, help develop theoretical models or conceptual frameworks, formulate mathematical claims, provide critical ingredients for proving mathematical claims, assist in the writing of proofs, assist with translation, clean and reformat datasets, or support qualitative and thematic data analysis does not apply to this work. Additionally, we used generative AI tools to create or modify scientific figures, including tables and images, sourcing/searching for information, edit the research paper to improve readability and quality of writing, identify relevant literature, and format references. We have reviewed all AI-assisted work, including text, code and figures; we have also verified the contents of the papers that were suggested by the LLMs as relevant work. We take responsibility for the final content of this work, including text, claims or artefacts produced with the aid of generative AI.

### ETHICS STATEMENT

Our experiments are conducted in simulated cooperative games and involve no direct human-participant studies or collection of new human gameplay data. Coding agents execute generated code in isolated workspaces without internet access. Compatibility with human-proxy and convention-following bots should not be interpreted as evidence of safe or effective coordination with humans in real-world settings. Deployment beyond these benchmarks would require additional evaluation to establish reliable and safe behaviour with human partners.

### REPRODUCIBILITY STATEMENT

To support reproducibility, we provide code for policy generation, evaluation, and analysis, together with the final generated policies and workspace snapshots. These artefacts allow the released policies to be evaluated without repeating policy generation. Section 4 describes the policy-generation setup, Appendix A provides the objective and selection instructions, and Section 5 details the experimental configuration and computational resources. Evaluation sample sizes and uncertainty measures are specified in the corresponding table and figure captions.

## REFERENCES

Saaket Agashe, Yue Fan, Anthony Reyna, and Xin Eric Wang. LLM-coordination: Evaluating and analyzing multi-agent coordination abilities in large language models. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 8053–8072. Association for Computational Linguistics, 2025.

Greg Anderson, Abhinav Verma, Isil Dillig, and Swarat Chaudhuri. Neurosymbolic reinforcement learning with formally verified exploration. In *Advances in Neural Information Processing Systems*, volume 33, 2020.

Artificial Analysis. Artificial analysis intelligence benchmarking methodology, n.d. URL `https://artificialanalysis.ai/methodology/intelligence-benchmarking`. Intelligence Index v4.3.2. Accessed September 26, 2026.

Nolan Bard, Jakob N. Foerster, Sarath Chandar, Neil Burch, Marc Lanctot, H. Francis Song, Emilio Parisotto, Vincent Dumoulin, Subhodeep Moitra, Edward Hughes, Iain Dunning, Shibl Mourad, Hugo Larochelle, Marc G. Bellemare, and Michael Bowling. The Hanabi challenge: A new frontier for AI research. *Artificial Intelligence*, 280:103216, 2020.

Daniel S. Bernstein, Robert Givan, Neil Immerman, and Shlomo Zilberstein. The complexity of decentralized control of Markov decision processes. *Mathematics of Operations Research*, 27(4): 819–840, 2002. doi: 10.1287/moor.27.4.819.297.

Francois Bredell, Herman A. Engelbrecht, and J. C. Schoeman. Augmenting the action space with conventions to improve multi-agent cooperation in Hanabi. *Autonomous Agents and Multi-Agent Systems*, 39:28, 2025.

Sébastien Bubeck, Christian Coester, Ronen Eldan, Timothy Gowers, Yin Tat Lee, Alexandru Lupsasca, Mehtaab Sawhney, Robert Scherrer, Mark Sellke, Brian K. Spears, Derya Unutmaz, Kevin Weil, Steven Yin, and Nikita Zhivotovskiy. Early science acceleration experiments with GPT-5. *arXiv preprint arXiv:2511.16072*, 2025.

Micah Carroll, Rohin Shah, Mark K. Ho, Tom Griffiths, Sanjit Seshia, Pieter Abbeel, and Anca Dragan. On the utility of learning about humans for human-AI coordination. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

Brandon Cui, Hengyuan Hu, Andrei Lupu, Samuel Sokota, and Jakob Foerster. Off-team learning. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

Marco Cusumano-Towner, David Hafner, Alexander Hertzberg, Brody Huval, Aleksei Petrenko, Eugene Vinitsky, Erik Wijmans, Taylor W. Killian, Stuart Bowers, Ozan Sener, Philipp Kraehenbuehl, and Vladlen Koltun. Robust autonomy emerges from self-play. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 11710–11737. PMLR, 2025.

Allan Dafoe, Edward Hughes, Yoram Bachrach, Tantum Collins, Kevin R. McKee, Joel Z. Leibo, Kate Larson, and Thore Graepel. Open problems in cooperative AI. *arXiv preprint arXiv:2012.08630*, 2020.

Tin Dizdarević, Ravi Hammond, Tobias Gessler, Anisoara Calinescu, Jonathan Cook, Matteo Gallici, Andrei Lupu, and Jakob Nicolaus Foerster. Ad-hoc human-AI coordination challenge. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 13900–13937. PMLR, 2025.

Jakob Foerster, Ioannis Alexandros Assael, Nando de Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 29, 2016.

Jakob Foerster, Francis Song, Edward Hughes, Neil Burch, Iain Dunning, Shimon Whiteson, Matthew Botvinick, and Michael Bowling. Bayesian action decoder for deep multi-agent reinforcement learning. In *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pp. 1942–1951. PMLR, 2019.

Johannes Forkel, Constantin Ruhdorfer, Michael Beukman, Andreas Bulling, and Jakob Foerster. High entropy leads to symmetry-equivariant policies in Dec-POMDPs. *arXiv preprint arXiv:2511.22581*, 2026. Version 5.

Letian Fu, Justin Yu, Karim El-Refai, Ethan Kou, Haoru Xue, Huang Huang, Wenli Xiao, Guanzhi Wang, Dantong Niu, Fei-Fei Li, Guanya Shi, Jiajun Wu, Shankar Sastry, Yuke Zhu, Ken Goldberg, and Linxi "Jim" Fan. CaP-X: A framework for benchmarking and improving coding agents for robot manipulation. *arXiv preprint arXiv:2603.22435*, 2026.

Alexander L. Gaunt, Marc Brockschmidt, Nate Kushman, and Daniel Tarlow. Differentiable programs with neural libraries. In *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pp. 1213–1222. PMLR, 2017.

Tobias Gessler, Tin Dizdarevic, Anisoara Calinescu, Benjamin Ellis, Andrei Lupu, and Jakob N. Foerster. OvercookedV2: Rethinking Overcooked for zero-shot coordination. In *International Conference on Learning Representations*, 2025.

H-Group. H-Group Conventions, n.d. URL `https://hanabi.github.io/`. Accessed September 17, 2026.

Adrian Hayler, Shashank Reddy Chirra, Andrei Lupu, Johannes Forkel, Bidipta Sarkar, Siheng Feng, and Jakob Nicolaus Foerster. Zero-shot coordination among LLM agents. In *Workshop on Multi-Agent Learning and Its Opportunities in the Era of Generative AI*, 2026.

Hengyuan Hu and Jakob N. Foerster. Simplified action decoder for deep multi-agent reinforcement learning. In *International Conference on Learning Representations*, 2020.

Hengyuan Hu and Dorsa Sadigh. Language instructed reinforcement learning for human-AI coordination. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 13584–13598. PMLR, 2023.

Hengyuan Hu, Adam Lerer, Alex Peysakhovich, and Jakob Foerster. "Other-Play" for zero-shot coordination. In *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 4399–4410. PMLR, 2020.

Hengyuan Hu, Adam Lerer, Brandon Cui, Luis Pineda, Noam Brown, and Jakob Foerster. Off-belief learning. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 4369–4379. PMLR, 2021.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, 2025.

Thomas Kwa, Ben West, Joel Becker, Amy Deng, Katharyn Garcia, Max Hasin, Sami Jawhar, Megan Kinniment, Nate Rush, Sydney Von Arx, Ryan Bloom, Thomas Broadley, Haoxing Du, Brian Goodrich, Nikola Jurkovic, Luke Miles, Seraphina Nix, Tao Lin, Neev Parikh, David Rein, Lucas Jun Koba Sato, Hjalmar Wijk, Daniel Ziegler, Elizabeth Barnes, and Lawrence Chan. Measuring AI ability to complete long software tasks. In *Advances in Neural Information Processing Systems*, volume 38, Main Conference, pp. 92213–92266. Curran Associates, Inc., 2025.

Niklas Lauffer, Ameesh Shah, Micah Carroll, Sanjit A. Seshia, Stuart Russell, and Michael Dennis. Robust and diverse multi-agent learning via rational policy gradient. In *Advances in Neural Information Processing Systems*, volume 38, 2025.

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)*, pp. 9493–9500, 2023.

Runyu Lu, Yubo Wu, Ethan Kou, Letian Fu, Wenli Xiao, Ajay Mandlekar, Yinzhen Xu, Guanya Shi, Ken Goldberg, Ang Chen, Mosharaf Chowdhury, Yuke Zhu, Linxi "Jim" Fan, and Guanzhi Wang. ASPIRE: Agentic /Skills discovery for robotics. *arXiv preprint arXiv:2607.00272*, 2026.

Andrei Lupu, Brandon Cui, Hengyuan Hu, and Jakob Foerster. Trajectory diversity for zero-shot coordination. In *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 7204–7213. PMLR, 2021.

Patrick MacAlpine, Katie Genter, Samuel Barrett, and Peter Stone. The RoboCup 2013 drop-in player challenges: Experiments in ad hoc teamwork. In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2014.

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 *Advances in Neural Information Processing Systems*, volume 36, 2023.

Stefanos Nikolaidis, David Hsu, and Siddhartha Srinivasa. Human-robot mutual adaptation in collaborative tasks: Models and experiments. *The International Journal of Robotics Research*, 36 (5–7):618–634, 2017.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

OpenAI. The Hugging Face incident and the road ahead, August 2026a. URL https://openai.com/index/hugging-face-incident-and-the-road-ahead/. August 26, 2026.

OpenAI. On the Navier–Stokes Millennium Prize Problem, September 2026b. URL https://openai.com/index/navier-stokes-solution/. September 8, 2026.

OpenAI. Ten advances in mathematics and theoretical computer science. Technical report, OpenAI, August 2026c. Updated August 6, 2026.

Mahesh Ramesh, Kaousheik Jayakumar, Aswinkumar Ramkumar, Pavan Thodima, Aniket Rege, and Emmanouil-Vasileios Vlatakis-Gkaragkounis. Sparks of cooperative reasoning: LLMs as strategic Hanabi agents. *arXiv preprint arXiv:2601.18077*, 2026.

Constantin Ruhdorfer, Matteo Bortoletto, Johannes Forkel, Jakob Nicolaus Foerster, and Andreas Bulling. The Yōkai learning environment: Tracking beliefs over space and time. In *Proceedings of the Reinforcement Learning Conference*, 2026.

Alexander Rutherford, Benjamin Ellis, Matteo Gallici, Jonathan Cook, Andrei Lupu, Garar Ingvarsson, Timon Willi, Ravi Hammond, Akbir Khan, Christian Schroeder de Witt, Alexandra Souly, Saptarashmi Bandyopadhyay, Mikayel Samvelyan, Minqi Jiang, Robert Tjarko Lange, Shimon Whiteson, Bruno Lacerda, Nick Hawes, Tim Rocktäschel, Chris Lu, and Jakob Nicolaus Foerster. JaxMARL: Multi-agent RL environments and algorithms in JAX. In *Advances in Neural Information Processing Systems*, volume 37, 2024.

Dorsa Sadigh, Shankar Sastry, Sanjit A. Seshia, and Anca D. Dragan. Planning for autonomous cars that leverage effects on human actions. In *Proceedings of Robotics: Science and Systems*, 2016.

A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):210–229, 1959.

Bidipta Sarkar, Andy Shih, and Dorsa Sadigh. Diverse conventions for human-AI collaboration. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

SemiAnalysis. What's the better business model for an ai lab, subscription or api?, June 2026. URL https://x.com/SemiAnalysis_/status/2064815042374074396. June 10, 2026.

DJ Strouse, Kevin McKee, Matt Botvinick, Edward Hughes, and Richard Everett. Collaborating with humans without human data. In *Advances in Neural Information Processing Systems*, volume 34, 2021.

Gerald Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6(2):215–219, 1994.

will-hanabi-bot. hanabi-bot. GitHub repository, n.d. URL https://github.com/will-hanabi-bot/hanabi-bot. Accessed September 20, 2026.

Zhaofeng Wu, Linlu Qiu, Alexis Ross, Ekin Akyürek, Boyuan Chen, Bailin Wang, Najoung Kim, Jacob Andreas, and Yoon Kim. Reasoning or reciting? exploring the capabilities and limitations of language models through counterfactual tasks. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 1819–1862. Association for Computational Linguistics, 2024.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V. Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *International Conference on Learning Representations*, 2024.

Ceyao Zhang, Kaijie Yang, Siyi Hu, Zihao Wang, Guanghe Li, Yihang Sun, Cheng Zhang, Zhaowei Zhang, Anji Liu, Song-Chun Zhu, Xiaojun Chang, Junge Zhang, Feng Yin, Yitao Liang, and Yaodong Yang. ProAgent: Building proactive cooperative agents with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 17591–17599, 2024a.

Hugh Zhang, Jeff Da, Dean Lee, Vaughn Robinson, Catherine Wu, Will Song, Tiffany Zhao, Pranav Raja, Charlotte Zhuang, Dylan Slack, Qin Lyu, Sean Hendryx, Russell Kaplan, Michele (Mike) Lunati, and Summer Yue. A careful examination of large language model performance on grade school arithmetic. In *Advances in Neural Information Processing Systems*, volume 37, 2024b.

# A POLICY-GENERATION INSTRUCTIONS

**Objective**

Maximize expected cooperative episode return when either role of this joint policy is paired with the complementary role from an independently generated policy. Your policy will be paired with independently generated partners. It should coordinate with them without assuming they use the same strategy, role assignments, or communication conventions. Seek competent, convention-robust behavior that can coordinate from locally observable evidence. Self-play return is a useful competence diagnostic, but it is not the optimization target.

**Selection rule**

The primary evaluation metric is XP: cooperative episode return when this policy is paired with independently generated policies. You cannot measure XP within this isolated policy-search run; it is evaluated only after many independent policy-search runs have produced their policies. Use the available self-play rollouts to verify basic task competence and diagnose failures. Leave in `policy.py` the policy you judge most likely to achieve the highest expected cross-play return with independently generated partners. Use your analysis and available evaluation evidence to inform that judgment; the highest self-play score does not necessarily identify the best cross-play policy. Avoid relying on a private convention that succeeds only when both roles use this same policy.

Figure 6: Objective and selection instructions for Overcooked V2. Note that these instructions are paraphrased across independent runs while preserving their meaning.

**Objective**

Maximize expected cooperative episode return when your local policy is paired with an independently generated partner policy, evaluated in both player positions. Your policy will be paired with independently generated partners. It should coordinate with them without assuming they use the same strategy, role assignments, or communication conventions. Seek competent, convention-robust behavior that can coordinate from locally observable evidence. Self-play return is a useful competence diagnostic, but it is not the optimization target.

**Selection rule**

The primary evaluation metric is XP: cooperative episode return when this policy is paired with independently generated policies. You cannot measure XP within this isolated policy-search run; it is evaluated only after many independent policy-search runs have produced their policies. Use the available self-play rollouts to verify basic task competence and diagnose failures. Leave in `policy.py` the policy you judge most likely to achieve the highest expected cross-play return with independently generated partners. Use your analysis and available evaluation evidence to inform that judgment; the highest self-play score does not necessarily identify the best cross-play policy. Avoid relying on a private convention that succeeds only when both roles use this same policy.

Figure 7: Objective and selection instructions for regular Hanabi and all Yokai configurations. Note that these instructions are paraphrased across independent runs while preserving their meaning.

**Objective**

Maximize expected cooperative episode return when your policy is paired with independently generated policies that follow H-group Level 1 conventions, evaluated in both player positions. All partners receive the same pinned convention context. Seek competent play and robust coordination with other convention-following agents. Assume shared convention meanings, while allowing differences in aspects such as implementation, tactical choices, risk thresholds, and tie-breaking wherever the conventions leave choices open. Self-play return is a useful competence diagnostic, but the optimization target is cross-play performance with independently generated convention-following partners.

**Selection rule**

The primary evaluation metric is XP: cooperative episode return when this policy is paired with independently generated policies that follow H-group Level 1 conventions. You cannot measure XP within this isolated policy-search run; it is evaluated only after many independent policy-search runs have produced their policies. Use the available self-play rollouts to verify basic task competence and diagnose failures. Leave in `policy.py` the policy you judge most likely to achieve the highest expected cross-play return with independently generated policies that follow H-group Level 1 conventions. Use your analysis and available evaluation evidence to inform that judgment; the highest self-play score does not necessarily identify the best cross-play policy. Avoid relying on a private convention that succeeds only when both roles use this same policy.

Figure 8: Objective and selection instructions for Hanabi with H-group Level 1 conventions. Note that these instructions are paraphrased across independent runs while preserving their meaning.

**Objective**

Maximize only the expected self-play episode return.

**Selection rule**

Leave in `policy.py` the policy with the highest authoritative development mean self-play episode return. Use minimum return and return variance only to diagnose or break ties between candidates with the same mean return.

Figure 9: Self-play objective and selection instructions for Overcooked V2. Note that these instructions are paraphrased across independent runs while preserving their meaning.

## B FULL OVERCOOKED V2 RESULTS

Table 4: **Coordination performance on Overcooked V2.** Mean episode return, with ± SD shown below each mean. Coding-agent results use six seeds and 1,024 games per policy pairing; published baselines, taken directly from Gessler et al. (2025), use ten seeds and 500 games per pairing.

| | Demo Cook | | | | Grounded Coord. | | | | Test-time | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Simple | | Wide | | Simple | | Ring | | Simple | | Wide | |
| Algorithm $\mathcal{L}$ | SP | IXP | SP | IXP | SP | IXP | SP | IXP | SP | IXP | SP | IXP |
| Self-Play | 172 ± 19 | 36 ± 82 | 195 ± 4 | 40 ± 61 | 154 ± 10 | −55 ± 124 | 164 ± 10 | −12 ± 75 | 145 ± 22 | −81 ± 99 | 194 ± 12 | −30 ± 96 |
| State-Augmented | 170 ± 16 | 112 ± 50 | 192 ± 7 | 60 ± 60 | 156 ± 29 | −17 ± 115 | 165 ± 7 | −16 ± 65 | 161 ± 18 | −55 ± 103 | 137 ± 82 | −12 ± 72 |
| Other-Play | 149 ± 25 | 47 ± 48 | 162 ± 23 | 62 ± 53 | 122 ± 24 | 4 ± 73 | 121 ± 36 | −9 ± 21 | 121 ± 37 | −3 ± 51 | 175 ± 54 | −11 ± 42 |
| Fictitious Co-Play | 22 ± 48 | 1 ± 25 | 45 ± 43 | 13 ± 28 | 64 ± 58 | 46 ± 47 | 86 ± 34 | 6 ± 46 | 35 ± 44 | 6 ± 29 | 95 ± 56 | 23 ± 40 |
| GPT-5.6 Luna | 77 ± 41 | 85 ± 10 | 112 ± 10 | 110 ± 6 | 116 ± 19 | 82 ± 13 | 105 ± 15 | 105 ± 6 | 54 ± 59 | 8 ± 27 | 67 ± 58 | 4 ± 16 |
| Claude Sonnet 5 | 125 ± 22 | 101 ± 19 | 103 ± 45 | 91 ± 17 | 132 ± 13 | 130 ± 5 | 104 ± 51 | 101 ± 20 | 76 ± 42 | 57 ± 2 | 113 ± 31 | 73 ± 25 |
| GPT-5.5 | 150 ± 11 | 125 ± 10 | 153 ± 17 | 133 ± 18 | 137 ± 8 | 101 ± 3 | 122 ± 12 | 122 ± 5 | 127 ± 77 | 33 ± 47 | 195 ± 17 | 195 ± 7 |
| GPT-5.6 Sol | 207 ± 16 | 174 ± 27 | 185 ± 32 | 162 ± 29 | 149 ± 18 | 129 ± 9 | 152 ± 17 | 133 ± 14 | 180 ± 1 | 155 ± 19 | 214 ± 17 | 202 ± 11 |
| Claude Opus 5 | 198 ± 4 | 197 ± 1 | 202 ± 4 | 193 ± 8 | 139 ± 8 | 130 ± 5 | 147 ± 4 | 130 ± 4 | 172 ± 6 | 168 ± 2 | 209 ± 11 | 120 ± 26 |
| GPT-6 Astra | 198 ± 3 | 191 ± 7 | 213 ± 9 | 211 ± 4 | 138 ± 4 | 137 ± 1 | 152 ± 3 | 150 ± 2 | 161 ± 12 | 155 ± 5 | 214 ± 11 | 177 ± 26 |
| Claude Fable 5.1 | 200 ± 11 | 196 ± 1 | 212 ± 16 | 203 ± 5 | 134 ± 3 | 134 ± 1 | 158 ± 16 | 130 ± 7 | 174 ± 9 | 172 ± 4 | 231 ± 10 | 218 ± 15 |

Table 5: **Coordination performance on novel Overcooked V2 layouts.** Mean episode return, with ± SD shown below each mean. Coding-agent results use six seeds and 1,024 games per policy pairing.

| | Block Grid | | Pipeline Cook | |
|---|---|---|---|---|
| Algorithm $\mathcal{L}$ | SP | IXP | SP | IXP |
| GPT-6 Sol | 51 ± 14 | 45 ± 5 | 64 ± 6 | 59 ± 5 |
| Claude Opus 5.5 | 67 ± 8 | 69 ± 3 | 77 ± 3 | 73 ± 1 |
| GPT-6 Astra | 74 ± 12 | 66 ± 6 | 81 ± 3 | 77 ± 3 |
| Claude Fable 5.1 | 76 ± 12 | 64 ± 4 | 80 ± 2 | 75 ± 3 |

## C FULL YOKAI RESULTS

Table 6: **Coordination performance on two-player Yokai.** Mean episode return ± SEM. Coding-agent results use six seeds and 1,024 games per policy pairing; the 12-seed method baselines are taken directly from Ruhdorfer et al. (2026), who evaluated them on 5,000 games. They also report an illustrative human-human baseline of $5.7 \pm 0.5$ across 25 games in the nine-card imperfect-memory setting.

| Configuration | Algorithm $\mathcal{L}$ | SP | IXP |
|---|---|---|---|
| 9 cards Perfect memory | IPPO | 7.5 ± 0.3 | 1.6 ± 0.2 |
| | IPPO+High Ent. | 7.5 ± 0.1 | 4.1 ± 0.8 |
| | Other-Play | 6.6 ± 0.2 | 4.8 ± 0.1 |
| | OBL L1 | 4.0 ± 0.2 | 2.3 ± 0.3 |
| | OBL L2 | 4.3 ± 0.2 | 2.1 ± 0.2 |
| | OBL L3 | 4.4 ± 0.4 | 2.2 ± 0.3 |
| | OBL L4 | 4.5 ± 0.2 | 2.8 ± 0.3 |
| | OBL L5 | 4.8 ± 0.2 | 2.5 ± 0.3 |
| | GPT-5.6 Luna | 1.5 ± 0.5 | 0.7 ± 0.1 |
| | Claude Sonnet 5 | 3.5 ± 0.1 | 3.6 ± 0.0 |
| | GPT-5.5 | 3.4 ± 0.2 | 2.0 ± 0.3 |
| | GPT-5.6 Sol | 4.9 ± 0.3 | 4.7 ± 0.1 |
| | Claude Opus 5 | 6.0 ± 0.5 | 5.6 ± 0.3 |
| | GPT-6 Astra | 7.5 ± 0.5 | 7.3 ± 0.2 |
| | Claude Fable 5.1 | 7.0 ± 0.3 | 7.0 ± 0.1 |
| 9 cards Imperfect memory | Other-Play | 0.0 ± 0.0 | 0.0 ± 0.0 |
| | Other-Play (TrXL) | 0.3 ± 0.1 | 0.3 ± 0.1 |
| | GPT-5.6 Luna | 0.7 ± 0.4 | 0.3 ± 0.1 |
| | Claude Sonnet 5 | 3.5 ± 0.1 | 3.4 ± 0.1 |
| | GPT-5.5 | 3.7 ± 0.2 | 2.6 ± 0.1 |
| | GPT-5.6 Sol | 4.9 ± 0.5 | 4.3 ± 0.1 |
| | Claude Opus 5 | 5.2 ± 0.5 | 5.2 ± 0.2 |
| | GPT-6 Astra | 6.3 ± 0.5 | 6.5 ± 0.2 |
| | Claude Fable 5.1 | 7.1 ± 0.3 | 6.9 ± 0.2 |
| 16 cards Perfect memory | IPPO | 0.0 ± 0.0 | 0.0 ± 0.0 |
| | GPT-5.6 Luna | 0.4 ± 0.2 | 0.0 ± 0.0 |
| | Claude Sonnet 5 | 1.8 ± 0.7 | 0.8 ± 0.1 |
| | GPT-5.5 | 2.2 ± 0.4 | 0.4 ± 0.1 |
| | GPT-5.6 Sol | 6.3 ± 0.4 | 5.1 ± 0.2 |
| | Claude Opus 5 | 6.1 ± 0.6 | 4.2 ± 0.7 |
| | GPT-6 Astra | 8.2 ± 0.9 | 8.0 ± 0.4 |
| | Claude Fable 5.1 | 8.4 ± 0.2 | 8.0 ± 0.1 |

# D FULL HANABI RESULTS

Table 7: **Coordination performance on two-player Hanabi.** Mean ± SEM over six coding-agent runs and four checkpoints per baseline, with 1,024 games per policy pairing or policy-bot seat assignment. We evaluate R2D2-based OBL checkpoints from Hu et al. (2021) and IPPO, IPPO+HE, and IPPO-based OBL checkpoints from Forkel et al. (2026).

| Algorithm $\mathcal{L}$ | Setting | SP | IXP | Human Proxy Bot XP (SP: 23.80) | Wilflame Bot XP (SP: 21.05) |
|---|---|---|---|---|---|
| IPPO | Standard | 24.53 ± 0.02 | 15.63 ± 2.13 | 13.60 ± 1.54 | 3.96 ± 0.48 |
| IPPO+HE | Standard | 24.48 ± 0.01 | 24.43 ± 0.02 | 11.64 ± 0.09 | 4.41 ± 0.08 |
| OBL L4 (R2D2) | Standard | 24.12 ± 0.03 | 23.76 ± 0.08 | 19.70 ± 0.16 | 5.19 ± 0.04 |
| OBL L6 (IPPO) | Standard | 24.35 ± 0.01 | 24.32 ± 0.00 | 14.92 ± 0.15 | 5.56 ± 0.13 |
| GPT-5.6 Luna | Standard | 13.82 ± 1.05 | 13.19 ± 0.54 | 4.88 ± 1.56 | 1.80 ± 0.56 |
| | H-group L1 | 1.54 ± 1.01 | 0.93 ± 0.28 | 1.36 ± 0.56 | 1.28 ± 0.36 |
| Claude Sonnet 5 | Standard | 16.80 ± 1.22 | 15.60 ± 0.61 | 11.08 ± 3.23 | 6.02 ± 1.73 |
| | H-group L1 | 11.47 ± 1.40 | 10.49 ± 0.87 | 13.51 ± 1.47 | 12.79 ± 0.81 |
| GPT-5.5 | Standard | 16.76 ± 0.76 | 12.97 ± 0.85 | 13.68 ± 1.79 | 7.13 ± 1.16 |
| | H-group L1 | 11.40 ± 2.45 | 7.42 ± 1.09 | 8.88 ± 1.89 | 10.05 ± 1.39 |
| GPT-5.6 Sol | Standard | 21.93 ± 0.19 | 18.74 ± 0.42 | 15.53 ± 2.25 | 7.42 ± 1.15 |
| | H-group L1 | 18.59 ± 0.65 | 18.26 ± 0.29 | 16.92 ± 0.57 | 18.54 ± 0.47 |
| Claude Opus 5 | Standard | 20.00 ± 0.36 | 19.35 ± 0.12 | 18.60 ± 0.36 | 12.38 ± 0.63 |
| | H-group L1 | 20.05 ± 0.42 | 19.09 ± 0.20 | 17.75 ± 0.24 | 18.21 ± 0.63 |
| GPT-6 Astra | Standard | 19.69 ± 0.80 | 19.43 ± 0.35 | 17.02 ± 1.61 | 12.09 ± 1.55 |
| | H-group L1 | 21.53 ± 0.34 | 21.11 ± 0.19 | 17.87 ± 0.33 | 19.98 ± 0.25 |
| Claude Fable 5.1 | Standard | 20.55 ± 0.26 | 19.50 ± 0.16 | 19.34 ± 0.68 | 14.87 ± 0.99 |
| | H-group L1 | 21.81 ± 0.21 | 20.80 ± 0.12 | 18.17 ± 0.40 | 20.25 ± 0.28 |

# E INTER-MODEL CROSS-PLAY MATRICES

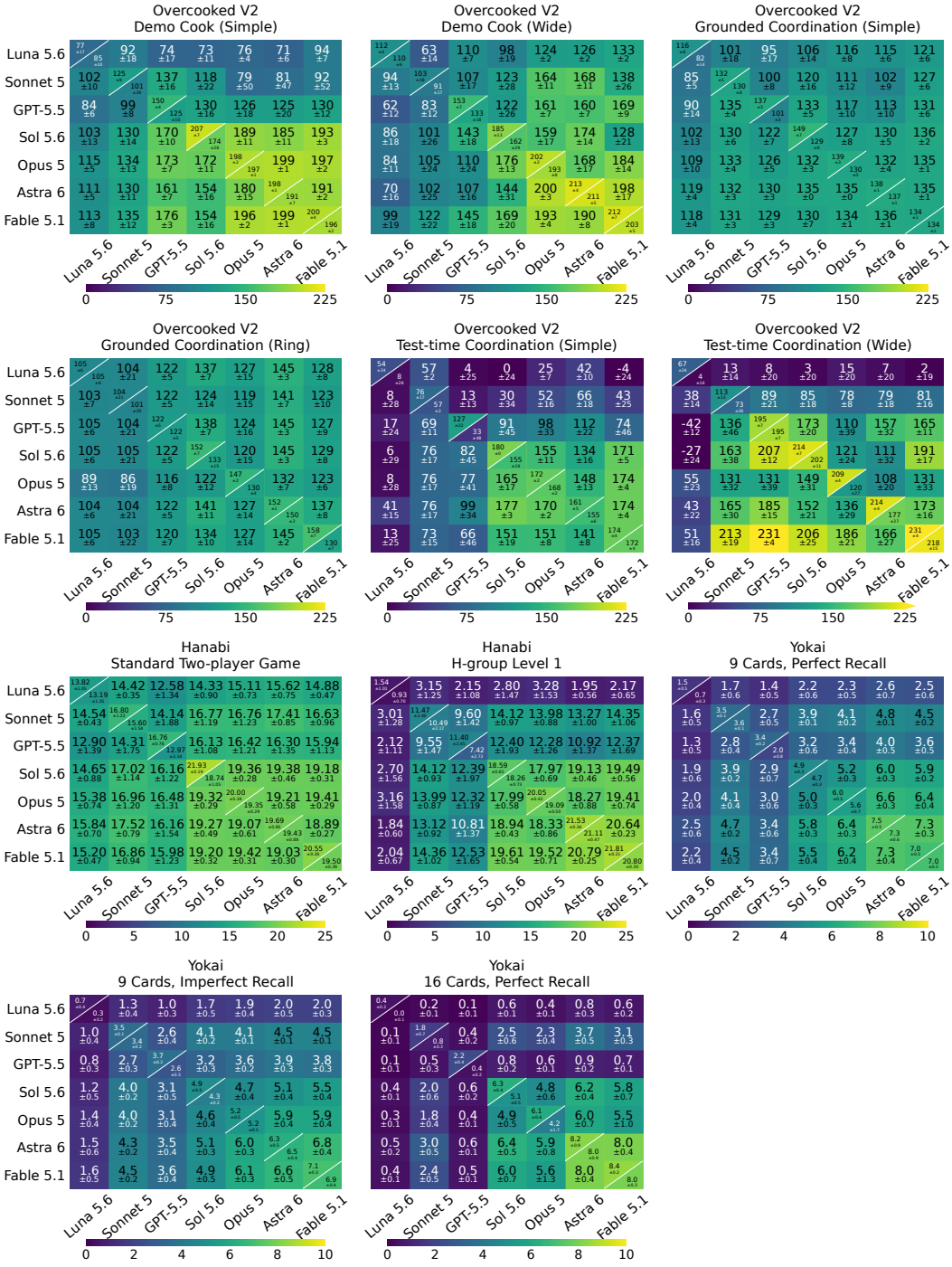



Figure 10: **Inter-model cross-play across all eleven environments.** Off-diagonal cells show inter-model XP; split diagonal cells show same-policy SP (upper-left) and within-model IXP (lower-right). Values report mean ± SEM using the evaluation procedure in Figure 4. For SP, SEM is computed across six seed-level scores; for IXP and inter-model XP, it is estimated using a delete-one-seed-block jackknife.

# F LEARNING CURVES

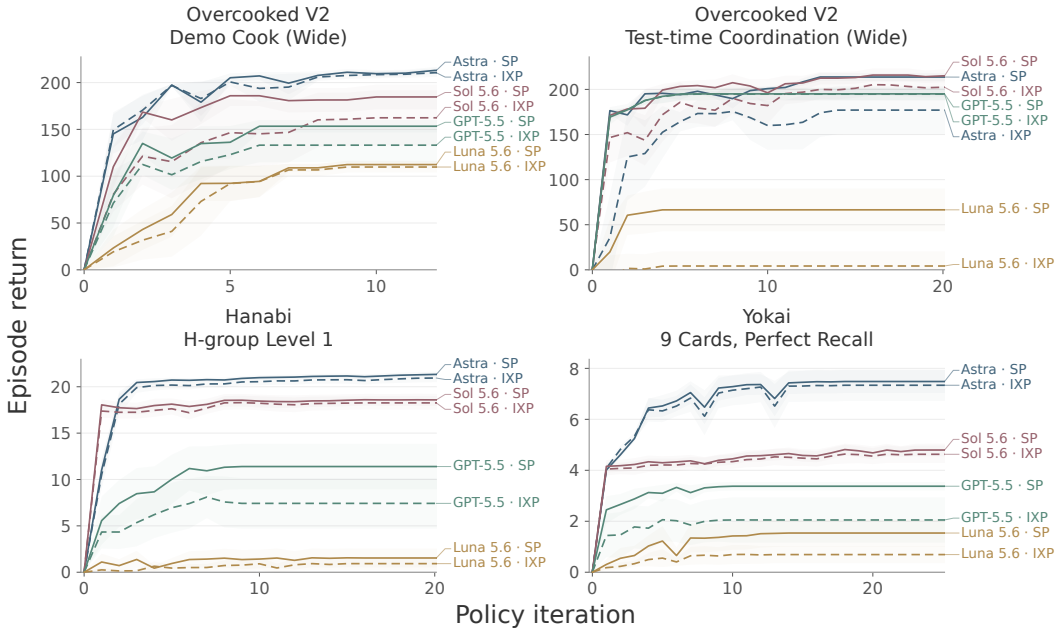



Figure 11: **Learning curves of coding agents.** Curves show mean performance across six seeds, with shaded regions indicating ± SEM. Runs that terminate early retain their final value through the maximum number of attempts observed, which can produce flat tails.

# G EXAMPLES OF GENERATED POLICIES

## G.1 GPT-6 ASTRA ON OVERCOOKED V2: GROUNDED COORDINATION (SIMPLE)

Adapts recipe-signal timing and uses displayed or inferred recipes to guide cooking.

```python
"""Grounded recipe communication and local, observation-driven cooking."""

import jax.numpy as jnp

POLICY_API_VERSION = 1
POLICY_METADATA = {'name': 'grounded_cook', 'adapter': 'overcooked_v2', 'adapter_version': 1, 'max_height': 5, 'max_width': 8, 'num_ingredients': 3}


def locate(obs, channel, global_row, global_col):
    index = jnp.argmax(obs[:, :, channel].reshape(-1))
    return global_row + 2 - index // 5, global_col + 2 - index % 5


def approach(row, col, facing, target_row, target_col, direction):
    move = jnp.where(row < target_row, 1, jnp.where(row > target_row, 3,
           jnp.where(col < target_col, 0, jnp.where(col > target_col, 2, 4))))
    turn = jnp.asarray([3, 1, 0, 2], jnp.int32)[direction]
    return jnp.where((row == target_row) & (col == target_col),
                     jnp.where(facing == direction, 5, turn), move).astype(jnp.int32)


def init_agent_0():
    return jnp.asarray([0, 0, 100, 0, 0, -1, 0, 6, 0], jnp.int32)


def agent_0_act(obs, reset, memory, rng):
    memory = jnp.where(reset, init_agent_0(), memory)
    time, old_count, since_signal, empty_age, retries, old_recipe, last_fill, fill_gap, warned = memory
    row, col = locate(obs, 23, 2, 0)
    facing = jnp.argmax(obs[2, 2, 1:5])
    pot = jnp.sum(obs[:, :, 29:34] * obs[:, :, 22:23], axis=(0, 1))
    count = jnp.sum(pot[2:])
    cleared = (old_count > 0) & (count == 0) & (col == 2)
    inserted = (count > old_count) & (col == 2)
    fill_gap = jnp.where(inserted & (old_count > 0), jnp.clip(time - last_fill, 6, 24), fill_gap)
    last_fill = jnp.where(inserted, time, last_fill)
    warned = jnp.where(cleared, 0, warned)
    empty_age = jnp.where(cleared, 0, empty_age + 1)
    since_signal = jnp.where(cleared, 100, since_signal + 1)
    retries = jnp.where(cleared, 0, retries)
    recipe = jnp.argmax(jnp.sum(obs[:, :, 36:39] * obs[:, :, 23:24], axis=(0, 1)))
    changed = (old_recipe >= 0) & (recipe != old_recipe)
    display_active = jnp.sum(obs[:, :, 36:39] * obs[:, :, 24:25]) > 0
    retry_base = jnp.maximum(20, (3 * fill_gap) // 2)
    retry_interval = retry_base * jnp.left_shift(jnp.asarray(1, jnp.int32), jnp.minimum(jnp.maximum(retries - 1, 0), 3))
    signal_delay = jnp.maximum(3, (2 * fill_gap) // 3 - 5)
    next_cycle_steps = 20 + (11 * fill_gap) // 3
    cycle_signal = (count == 0) & (((empty_age >= signal_delay) & (since_signal >= retry_interval)) | changed)
    correction = (count > 0) & (jnp.argmax(pot[2:]) != recipe) & (warned == 0) & (since_signal >= 10)
    ready = (cycle_signal | correction) & ~display_active & (time + next_cycle_steps < 400)
    action = approach(row, col, facing, 2, 2, 2)
    at_button = (row == 2) & (col == 2) & (facing == 2)
    action = jnp.where(at_button & ~ready, 4, action).astype(jnp.int32)
    activated = at_button & ready
    since_signal = jnp.where(activated, 0, since_signal)
    warned = jnp.where(activated & correction, 1, warned)
    retries = retries + activated.astype(jnp.int32)
    memory = jnp.stack([time + 1, jnp.where(col == 2, count, old_count), since_signal, empty_age, retries, recipe, last_fill, fill_gap, warned]).astype(jnp.int32)
    return action, memory, {'pot_count': count, 'activated': activated, 'empty_age': empty_age, 'fill_gap': fill_gap}


def counter_options(obs, row, col, facing, recipe, source):
    # All four usable counters are directly observable from the middle row.
    counter_rows = jnp.asarray([0, 0, 4, 4], jnp.int32)
    counter_cols = jnp.asarray([5, 6, 5, 6], jnp.int32)
    local_rows, local_cols = counter_rows - row + 2, counter_cols - col + 2
    seen = (local_rows >= 0) & (local_rows < 5) & (local_cols >= 0) & (local_cols < 5)
    items = obs[jnp.clip(local_rows, 0, 4), jnp.clip(local_cols, 0, 4), 29:34]
    standing_rows = jnp.where(counter_rows == 0, 1, 3)
```

```
    directions = jnp.where(counter_rows == 0, 0, 1)
    distance = jnp.abs(row - standing_rows) + jnp.abs(col - counter_cols)
    free = seen & (jnp.sum(items, axis=1) == 0)
    soup = seen & (items[:, 1] > 0) & (items[:, 2 + recipe] == 3)
    raw = seen & (items[:, 0] == 0) & (items[:, 1] == 0) & (items[:, 2 + source] == 1)
    def select(mask):
        index = jnp.argmin(jnp.where(mask, distance, 100))
        action = approach(row, col, facing, standing_rows[index], counter_cols[index], directions[index])
        return jnp.any(mask), action
    return select(free), select(soup), select(raw)


def init_agent_1():
    # time, recipe, confidence (0 unknown / 1 displayed / 2 inferred),
    # previous goal interaction, wait count, ever informed, silent cycles
    return jnp.asarray([0, 0, 0, 0, 0, 0, 0], jnp.int32)


def agent_1_act(obs, reset, memory, rng):
    memory = jnp.where(reset, init_agent_1(), memory)
    time, recipe, valid, old_goal, waiting, informed, silent_cycles = memory
    row, col = locate(obs, 22, 2, 4)
    facing = jnp.argmax(obs[2, 2, 1:5])
    inv = obs[2, 2, 5:10]
    dish, plate = inv[1] > 0, inv[0] > 0
    raw = (jnp.sum(inv[2:]) > 0) & ~dish
    empty = jnp.sum(inv) == 0
    delivered = (old_goal > 0) & ~dish
    valid = jnp.where(delivered, 0, valid)
    waiting = jnp.where(delivered, 0, waiting)

    displayed = jnp.sum(obs[:, :, 36:39], axis=(0, 1))
    visible = jnp.sum(displayed) > 0
    recipe = jnp.where(visible, jnp.argmax(displayed), recipe)
    valid = jnp.where(visible, 1, valid)
    informed = jnp.maximum(informed, visible.astype(jnp.int32))
    silent_cycles = jnp.where(visible, 0, silent_cycles)
    waiting = jnp.where((valid == 0) & empty, waiting + 1, waiting)
    wait_limit = jnp.where(time < 40, 20, jnp.where((informed == 0) | (silent_cycles >= 3), 4, 10))
    fallback = (valid == 0) & (waiting >= wait_limit)
    silent_cycles = silent_cycles + fallback.astype(jnp.int32)
    recipe = jnp.where(fallback & ((informed == 0) | (silent_cycles >= 3)), 1 - recipe, recipe)
    valid = jnp.where(fallback, 2, valid)

    pot = jnp.sum(obs[:, :, 29:34] * obs[:, :, 22:23], axis=(0, 1))
    count = jnp.sum(pot[2:])
    cooked = pot[1] > 0
    # A partly filled pot cannot be undone: finish it homogeneously, then
    # store it if a later grounded display says the dish would be wrong.
    source = jnp.where(count > 0, jnp.argmax(pot[2:]), recipe)
    source_row = jnp.where(source == 0, 1, 3)
    get_raw = approach(row, col, facing, source_row, 6, 2)
    pot_action = approach(row, col, facing, 2, 5, 3)
    get_plate = approach(row, col, facing, jnp.where(row == 3, 3, 1), 5, 3)
    deliver = approach(row, col, facing, 2, 6, 2)
    wait_display = jnp.where(pot_action == 5, 4, pot_action)

    (free_counter, store_action), (stored_soup, soup_action), (stored_raw, raw_action) = counter_options(obs, row, col, facing, recipe, source)
    get_raw = jnp.where(stored_raw, raw_action, get_raw)
    acquire = jnp.where((count == 0) & stored_soup, soup_action, get_raw)
    mismatched = jnp.argmax(inv[2:]) != recipe
    park_dish = dish & mismatched & (valid == 1) & free_counter
    park_raw = raw & mismatched & (valid == 1) & (count == 0) & free_counter

    action = jnp.where(dish, jnp.where(park_dish, store_action, deliver),
             jnp.where(plate, jnp.where(cooked, pot_action, jnp.where(pot_action == 5, 4, pot_action)),
             jnp.where(raw, jnp.where(park_raw, store_action, pot_action),
             jnp.where(count >= 3, get_plate,
             jnp.where((valid > 0) | (count > 0), acquire, wait_display)))))
    goal_attempt = dish & (row == 2) & (col == 6) & (facing == 2) & (action == 5)
    memory = jnp.stack([time + 1, recipe, valid, goal_attempt.astype(jnp.int32), waiting, informed, silent_cycles]).astype(jnp.int32)
    trace = {'recipe': recipe, 'valid': valid, 'pot_count': count,
             'waiting': waiting, 'delivered': delivered, 'row': row, 'col': col,
             'silent_cycles': silent_cycles, 'park_dish': park_dish,
             'park_raw': park_raw, 'stored_soup': stored_soup}
    return action.astype(jnp.int32), memory, trace
```

### G.2 GPT-6 ASTRA ON YOKAI: SIXTEEN CARDS, PERFECT MEMORY

Updates beliefs from partner relocations to guide card observations and moves.

```python
"""Local evidence and geometric color grouping for cooperative Yokai."""

import jax
import jax.numpy as jnp

POLICY_API_VERSION = 1
POLICY_METADATA = {'name': 'bare_candidate', 'adapter': 'yokai', 'adapter_version': 1, 'num_agents': 2, 'num_colors': 4, 'deck_size': 16, 'grid_size': 10, 'num_hints': 7, 'num_nodes': 23, 'node_features': 15, 'num_actions': 2325, 'policy_structure': 'shared_local'}


def init_agent():
    return {"colors": jnp.zeros((16, 4), jnp.float32),
            "positions": jnp.zeros((16, 2), jnp.int32),
            "relations": jnp.zeros((16, 16), jnp.float32),
            "future": jnp.zeros((16,), jnp.bool_)}


def beliefs(obs, seen):
    hints = obs.node_features[16:, 4:8].astype(jnp.float32)
    attached = obs.adjacency_matrix[:16, 16:].astype(jnp.float32)
    support = attached @ hints
    allowed = jnp.where(jnp.sum(support, axis=1, keepdims=True) > 0, support, 1.)
    known = jnp.sum(seen, axis=1, keepdims=True) > 0
    domains = jnp.where(known, seen > 0, allowed > 0)
    def force(_, d):
        singleton = jnp.sum(d, axis=1, keepdims=True) == 1
        fixed_count = jnp.sum(d & singleton, axis=0, keepdims=True)
        possible_count = jnp.sum(d, axis=0, keepdims=True)
        d = d & ((fixed_count < 4) | singleton)
        forced = d & (possible_count == 4)
        d = jnp.where(jnp.any(forced, axis=1, keepdims=True), forced, d)
        return d
    domains = jax.lax.fori_loop(0, 16, force, domains)
    # Fall back to own observations if an independently supplied hint conflicts.
    domains = jnp.where(jnp.any(domains, axis=1, keepdims=True), domains, jnp.where(known, seen > 0, True))
    known = jnp.sum(domains, axis=1, keepdims=True) == 1
    seen = domains.astype(jnp.float32)
    p = seen / jnp.sum(seen, axis=1, keepdims=True)
    def balance(_, p):
        remain = jnp.maximum(4. - jnp.sum(jnp.where(known, seen, 0.), axis=0), 0.)
        u = jnp.where(known, 0., p)
        u = u * remain / jnp.maximum(jnp.sum(u, axis=0), 1e-6)
        u = u / jnp.maximum(jnp.sum(u, axis=1, keepdims=True), 1e-6)
        return jnp.where(known, seen, u)
    return jax.lax.fori_loop(0, 8, balance, p)


def removed_components(edge):
    ids = jnp.arange(16)
    present = ids[None, :] != ids[:, None]
    edges = edge[None] & present[:, :, None] & present[:, None, :]
    labels = jnp.broadcast_to(ids, (16, 16))
    def propagate(_, lab):
        return jnp.minimum(lab, jnp.min(jnp.where(edges, lab[:, None, :], 16), axis=-1))
    labels = jax.lax.fori_loop(0, 15, propagate, labels)
    reach = (labels[:, :, None] == labels[:, None, :]) & present[:, :, None] & present[:, None, :]
    reps = (labels == ids[None, :]) & present
    return reach, reps


def motion_beliefs(p, relations):
    fixed = jnp.max(p, axis=1, keepdims=True) > .999
    remaining = jnp.maximum(4. - jnp.sum(jnp.where(fixed, p, 0.), axis=0), 0.)
    def update(_, q):
        q = p * jnp.exp(3. * (relations @ q))
        q = q / jnp.maximum(jnp.sum(q, axis=1, keepdims=True), 1e-6)
        u = jnp.where(fixed, 0., q)
        u = u * remaining / jnp.maximum(jnp.sum(u, axis=0), 1e-6)
        u = u / jnp.maximum(jnp.sum(u, axis=1, keepdims=True), 1e-6)
        return jnp.where(fixed, p, u)
    return jax.lax.fori_loop(0, 8, update, p)


def geometry(obs, p, relations, positions=None):
    ids = jnp.arange(16)
    present = ids[None, :] != ids[:, None]
    pos = obs.positions if positions is None else positions
```

```
    dist = jnp.sum(jnp.abs(pos[:, None] - pos[None, :]), axis=-1)
    certain = jnp.max(p, axis=1) > .999
    color = jnp.argmax(p, axis=1)
    edge = (dist == 1) & (color[:, None] == color[None, :]) & certain[:, None] & certain[None, :]
    # Compute the color components after removing each possible moving card.
    reach, reps = removed_components(edge)
    cells = jnp.stack((jnp.arange(100) // 10, jnp.arange(100) % 10), axis=1)
    cd = jnp.sum(jnp.abs(cells[:, None, :] - pos[None, :, :]), axis=-1)
    near = (cd == 1).astype(jnp.float32)
    touched = jnp.einsum('dj,ijk->idk', near, reach.astype(jnp.float32)) > 0
    oldtouched = jnp.einsum('ij,ijk->ik', (dist == 1).astype(jnp.float32), reach.astype(jnp.
    ↪ float32)) > 0
    same = p @ p.T
    confidence = jnp.where(certain, 1., .12 + .6 * jnp.maximum((jnp.max(p, axis=1) - .25) / .75,
    ↪ 0.))
    weights = same * confidence[:, None] * confidence[None, :]
    weights = jnp.where(certain[:, None] & certain[None, :], weights, jnp.maximum(weights,
    ↪ relations))
    comp_gain = jnp.sum((touched.astype(jnp.float32) - oldtouched[:, None, :]) * reps[:, None, :]
    ↪ * weights[:, None, :], axis=-1)
    edge_gain = jnp.einsum('dj,ij->id', near, weights * present) - jnp.sum((dist == 1) * weights,
    ↪ axis=1)[:, None]
    distance_gain = jnp.sum((dist[:, None, :] - cd[None]) * weights[:, None, :] * present[:, None,
    ↪  :], axis=-1) / jnp.maximum(jnp.sum(weights * present, axis=-1)[:, None], 1.)
    degree = jnp.sum(near, axis=1)[None, :] - near.T
    scores = 12. * comp_gain + .65 * edge_gain + .7 * distance_gain + .025 * degree
    fullreach, fullreps = removed_components(dist == 1)
    touch_count = jnp.einsum('dj,ijk->idk', near, fullreach.astype(jnp.float32))
    fulltouch = touch_count > 0
    vacant = ~jnp.any(cd == 0, axis=1)
    legal = vacant[None, :] & jnp.all(fulltouch | ~fullreps[:, None, :], axis=-1) & (obs.
    ↪ node_features[:16, 10:11] == 0)
    unrestricted = jnp.max(jnp.where(vacant[None, :], scores, -1e6), axis=1)
    constrained = jnp.max(jnp.where(legal, scores, -1e6), axis=1)
    blocked = (jnp.sum(fullreps, axis=1) > 1) & certain & (obs.node_features[:16, 10] == 0) & (
    ↪ constrained < 6.) & (unrestricted > 6.)
    bridge_all = jnp.all(fulltouch | ~fullreps[:, None, :], axis=-1)
    comp_size = jnp.sum(fullreach, axis=1)
    loses_contact = (near[None, :, :] > 0) & (touch_count == 1) & (comp_size[:, None, :] > 1)
    bridge = bridge_all[:, :, None] & ~loses_contact & present[:, None, :]
    bridge_value = jnp.max(bridge * (blocked * jnp.minimum(unrestricted - constrained, 12.))[:,
    ↪ None, None], axis=0).T
    raw_scores = jnp.where(legal, scores, -1e6)
    scores = jnp.where(legal, scores + .35 * bridge_value, -1e6)
    # Expected value of learning the moving card's color before choosing its destination.
    target = p * confidence[:, None]
    cg = jnp.einsum('idk,kc->idc', (touched.astype(jnp.float32) - oldtouched[:, None, :]) * reps
    ↪ [:, None, :], target)
    eg = jnp.einsum('idk,kc->idc', (near[None] - (dist == 1)[:, None, :]) * present[:, None, :],
    ↪ target)
    dg = jnp.einsum('idk,kc->idc', (dist[:, None, :] - cd[None]) * present[:, None, :], target)
    dg /= jnp.maximum(jnp.einsum('ik,kc->ic', present, target)[:, None, :], 1.)
    conditional = 12. * cg + .65 * eg + .7 * dg + .025 * degree[:, :, None]
    conditional = jnp.where(legal[:, :, None], conditional, -1e6)
    best_now = jnp.maximum(jnp.max(scores), 0.)
    info_value = jnp.sum(p * jnp.maximum(jnp.max(conditional, axis=1) - best_now, 0.), axis=1)
    return scores, edge, same, certain, info_value, raw_scores


def prepare_move(obs, p, relations, scores, raw_scores):
    _, first = jax.lax.top_k(scores.reshape(-1), 8)
    reveals = jnp.sum(jnp.any(obs.node_features[16:, 4:8] > 0, axis=1))
    placements = jnp.sum(obs.node_features[16:, 10])
    elapsed_turns = reveals + placements
    def follow(action):
        card, cell = action // 100, action % 100
        pos = obs.positions.at[card].set(jnp.stack((cell // 10, cell % 10)))
        adj = jnp.sum(jnp.abs(pos[:, None] - pos[None, :]), axis=-1) == 1
        future = geometry(obs, p, relations * adj, pos)[5]
        future = jnp.where((obs.node_features[:16, 11:12] == 1) | (elapsed_turns < 12), future, -1
    ↪ e6)
        best = jnp.argmax(future)
        return jnp.max(future), best // 100
    values, movers = jax.vmap(follow)(first)
    total = raw_scores.reshape(-1)[first] + .9 * values
    choice = jnp.argmax(total)
    useful = (elapsed_turns < 13) & (jnp.max(scores) < 8.) & (jnp.max(total) > jnp.max(raw_scores)
    ↪  + 4.) & (values[choice] > 6.)
    return first[choice], movers[choice], useful


def agent_act(obs, reset, memory, rng):
    memory = jax.tree.map(lambda x: jnp.where(reset, jnp.zeros_like(x), x), memory)
    features = obs.node_features
    visible = features[:16, :4].astype(jnp.float32)
```

```
    seen = jnp.where(jnp.sum(visible, axis=1, keepdims=True) > 0, visible, memory['colors'])
    p = beliefs(obs, seen)
    moved = jnp.any(obs.positions != memory['positions'], axis=1) & ~reset
    adj = jnp.sum(jnp.abs(obs.positions[:, None] - obs.positions[None, :]), axis=-1) == 1
    oldadj = jnp.sum(jnp.abs(memory['positions'][:, None] - memory['positions'][None, :]), axis
    =-1) == 1
    partner_seen = features[:16, 11] == 1
    suggested = moved[:, None] & adj & ~oldadj & partner_seen[:, None] & partner_seen[None, :] & (
    features[0, 12] == 0)
    evidence = .9 * suggested / jnp.maximum(jnp.sum(suggested, axis=1, keepdims=True), 1)
    relations = jnp.maximum(memory['relations'] * adj, evidence + evidence.T)
    planning = motion_beliefs(p, relations)
    scores, _, _, _, info_value, raw_scores = geometry(obs, planning, relations)
    certain = jnp.max(p, axis=1) > .999
    same = p @ p.T
    edge = adj & (same > .999) & certain[:, None] & certain[None, :]
    uncertainty = 1. - jnp.max(p, axis=1)
    # Gather fresh evidence, particularly about recent partner relocations.
    observe_score = uncertainty * (10. + 2. * moved + .2 * (1 - features[:16, 11]))
    observe_score += .8 * info_value * (~certain)
    observe_score += .01 * jnp.sum(obs.adjacency_matrix[:16, :16], axis=1)
    observe_action = 1 + jnp.argmax(jnp.where(obs.legal_actions[1:17], observe_score, -1e6))
    move_action = 17 + jnp.argmax(jnp.where(obs.legal_actions[17:1617].reshape(16, 100), scores,
    -1e6))
    planned_action, planned_mover, useful_plan = prepare_move(obs, planning, relations, scores,
    raw_scores)
    use_plan = useful_plan & (jnp.sum(certain) >= 12) & (features[0, 12] == 1) & (features[0, 13]
    == 2)
    move_action = jnp.where(use_plan, 17 + planned_action, move_action)
    future = jnp.where((features[0, 12] == 1) & (features[0, 13] == 2), (jnp.arange(16) ==
    planned_mover) & use_plan, memory['future'])
    hintset = features[16:, 4:8].astype(jnp.float32)
    match = hintset @ p.T
    reach = edge | jnp.eye(16, dtype=jnp.bool_)
    def closure(_, r):
        return (r.astype(jnp.int32) @ r.astype(jnp.int32)) > 0
    reach = jax.lax.fori_loop(0, 4, closure, reach)
    settled = jnp.sum(edge, axis=1).astype(jnp.float32)
    need_move = jnp.maximum(jnp.max(scores, axis=1), 0.)
    split_anchor = jnp.sum((same > .999) & ~reach & (features[:16, 10] == 1)[None, :], axis=1)
    hint_card_score = 100. * match + 3. * settled[None, :] - .5 * need_move[None, :] - 30. *
    split_anchor[None, :] - 30. * future[None, :]
    # Favor precise truthful public evidence when choosing among available hints.
    hint_card_score += (4. - jnp.sum(hintset, axis=1))[:, None] * 7.
    hint_card_score += (1. - features[:16, 11])[None, :] * (4. - jnp.sum(hintset, axis=1))[:, None
    ]
    at = obs.positions[:, 0] * 10 + obs.positions[:, 1]
    place_scores = jnp.full((7, 100), -1e6).at[:, at].set(hint_card_score)
    place_scores = jnp.where(obs.legal_actions[1624:2324].reshape(7, 100), place_scores, -1e6)
    place_action = 1624 + jnp.argmax(place_scores)
    reveal = obs.legal_actions[1617:1624]
    hint_size = jnp.sum(hintset, axis=1)[:, None]
    precise = (hint_size == 1) & (settled[None, :] >= 1.)
    useful_pair = (hint_size == 2) & (settled[None, :] >= 2.) & (features[:16, 11][None, :] == 0)
    & (split_anchor[None, :] == 0)
    inform = (precise | useful_pair) & (match > .999) & (need_move[None, :] < 6.)
    inform_scores = jnp.full((7, 100), -1e6).at[:, at].set(jnp.where(inform, hint_card_score, -1e6
    ))
    inform_scores = jnp.where(obs.legal_actions[1624:2324].reshape(7, 100), inform_scores, -1e6)
    can_inform = jnp.max(inform_scores) > -1e5
    hint_action = jnp.where(can_inform, 1624 + jnp.argmax(inform_scores), jnp.where(jnp.any(reveal
    ), 1617 + jnp.argmax(reveal), place_action))
    # A connected graph of each known color certifies success when all are known.
    solved = jnp.all(certain) & jnp.all((same < .999) | reach)
    substep = features[0, 13]
    action = jnp.where(substep < 2, observe_action, jnp.where(substep == 2, move_action,
    hint_action))
    action = jnp.where((substep == 0) & solved, 2324, action)
    action = jnp.where(features[0, 12] == 1, action, 0).astype(jnp.int32)
    future = jnp.where((features[0, 12] == 1) & (features[0, 13] == 3), False, future)
    memory = {'colors': seen, 'positions': obs.positions, 'relations': relations, 'future': future
    }
    return action, memory, {'known': jnp.sum(certain), 'solved': solved, 'move_value': jnp.max(
    scores), 'belief': p, 'planning': planning, 'positions': obs.positions, 'locked': features
    [:16, 10]}
```

### G.3 GPT-6 ASTRA ON HANABI: H-GROUP LEVEL 1

Interprets clues using play, save, and self-prompt alternatives.

```python
"""Two-player H-group Level 1 policy; all inference is observation-local."""
import jax
import jax.numpy as jnp

POLICY_API_VERSION = 1
POLICY_METADATA = {'name': 'h1_local_notes', 'adapter': 'hanabi', 'adapter_version': 1, 'num_agents': 2, 'num_colors': 5, 'num_ranks': 5, 'hand_size': 5, 'observation_size': 658, 'num_actions': 21, 'policy_structure': 'shared_local'}


def init_agent():
    return {'notes': jnp.ones((2, 5, 25), jnp.bool_),
            'previous': jnp.zeros(658, jnp.float32),
            'dependency': jnp.full((2, 5), -1, jnp.int32),
            'exempt': jnp.zeros((2, 5, 25), jnp.bool_),
            'wait': jnp.zeros((2, 5, 25, 5), jnp.bool_),
            'early': jnp.array(True)}


def decode(f):
    hand = f[:125].reshape(5, 25)
    k = f[308:].reshape(2, 5, 35)
    literal = k[:, :, :25] > 0
    touched = jnp.any(k[:, :, 25:] > 0, axis=-1)
    occupied = jnp.any(literal, axis=-1)
    heights = jnp.sum(f[167:192].reshape(5, 5) * jnp.arange(1, 6), axis=-1).astype(jnp.int32)
    d = f[203:253].reshape(5, 10)
    disc = jnp.stack([d[:, :3].sum(-1), d[:, 3:5].sum(-1), d[:, 5:7].sum(-1), d[:, 7:9].sum(-1), d[:, 9]], -1)
    counts = jnp.array([3, 2, 2, 2, 1])[None, :] - disc - (jnp.arange(5)[None, :] < heights[:, None])
    ranks = jnp.tile(jnp.arange(5), 5)
    colors = jnp.repeat(jnp.arange(5), 5)
    dead_suit = jnp.cumsum((counts <= 0) & (jnp.arange(5)[None, :] >= heights[:, None]), axis=1) > 0
    useful = ((jnp.arange(5)[None, :] >= heights[:, None]) & ~dead_suit).reshape(25)
    playable = ranks == heights[colors]
    return hand, literal, touched, occupied, heights, counts.reshape(25), useful, playable


def intersect_or(old, constraint, fallback):
    new = old & constraint
    return jnp.where(jnp.any(new, -1, keepdims=True), new, fallback)


def chop_of(touched, occupied):
    return jnp.min(jnp.where(~touched & occupied, jnp.arange(5), 5))


def focus_of(touch, touched, occupied):
    new = touch & ~touched
    chop = chop_of(touched, occupied)
    focus = jnp.where(jnp.any(new), jnp.max(jnp.where(new, jnp.arange(5), -1)), jnp.max(jnp.where(touch, jnp.arange(5), -1)))
    focus = jnp.where((chop < 5) & touch[jnp.minimum(chop, 4)], chop, focus)
    return jnp.maximum(focus, 0), chop, new


def tidy(notes, literal, touched, useful, visible, counts, heights, exempt):
    # Literal evidence always wins over a contradicted convention note.
    notes = intersect_or(notes, literal, literal)
    # Good Touch lets ambiguous clued cards shed played/unreachable identities.
    not_played = (jnp.tile(jnp.arange(5), 5) >= jnp.repeat(heights, 5))
    narrowed = notes & (not_played[None, None, :] | exempt)
    notes = jnp.where(touched[:, :, None] & jnp.any(narrowed, -1, keepdims=True), narrowed, notes)
    # Exact inferred cards cannot also be another useful clued card.
    def eliminate(_, n):
        exact = (n.sum(-1) == 1) & touched
        certain = n & exact[:, :, None]
        total = certain.sum(1)
        other = total[:, None, :] - certain > 0
        candidate = n & ~(other & touched[:, :, None] & ~exempt)
        n = jnp.where(jnp.any(candidate, -1, keepdims=True), candidate, n)
        return n
    notes = jax.lax.fori_loop(0, 2, eliminate, notes)
    # A visible, already-clued partner card is another Good Touch exclusion.
    partner_claim = jnp.any((visible > 0) & touched[1, :, None], axis=0)
    own = notes[0] & ~(partner_claim[None, :] & touched[0, :, None] & ~exempt[0])
    notes = notes.at[0].set(jnp.where(jnp.any(own, -1, keepdims=True), own, notes[0]))
```

```
    own_certain = jnp.any(notes[0] & touched[0, :, None] & (notes[0].sum(-1) == 1)[:, None], axis=0)
    partner = notes[1] & ~(own_certain[None, :] & touched[1, :, None] & ~exempt[1])
    notes = notes.at[1].set(jnp.where(jnp.any(partner, -1, keepdims=True), partner, notes[1]))
    return notes


def clue_notes(notes, literal_after, touched, occupied, receiver, touch, rank_clue, value, heights, counts, useful, other_cards):
    """Interpret one clue from its pre-clue hand, retaining play/save alternatives."""
    ranks = jnp.tile(jnp.arange(5), 5)
    colors = jnp.repeat(jnp.arange(5), 5)
    focus, chop, new = focus_of(touch, touched[receiver], occupied[receiver])
    recv = notes[receiver]
    other = 1 - receiver
    # Connectors must already be clued at the moment the clue is given.
    other_have = jnp.any(other_cards & touched[other, :, None], axis=0)
    exact_recv = recv & (recv.sum(-1) == 1)[:, None] & touched[receiver, :, None]
    current_playable = ranks == heights[colors]
    receiver_available = jnp.maximum(counts - other_cards.sum(0), 0)
    feasible_recv = recv & (receiver_available > 0)[None, :]
    ready_slots = touched[receiver] & jnp.any(feasible_recv, -1) & jnp.all(~feasible_recv | current_playable[None, :], -1)
    ready_cards = feasible_recv & ready_slots[:, None] & (jnp.arange(5) != focus)[:, None]
    own_have = jnp.any((exact_recv | ready_cards) & (jnp.arange(5) != focus)[:, None], axis=0)
    have = (other_have | own_have).reshape(5, 5)
    before = jnp.arange(5)[None, :] < heights[:, None]
    accounted = have | before
    connected = jnp.concatenate([jnp.ones((5, 1), jnp.bool_), jnp.cumprod(accounted[:, :4], axis=1).astype(jnp.bool_)], axis=1).reshape(25)
    play_set = connected & useful
    critical_save = (counts == 1) & (ranks != 4)
    special_save = rank_clue & ((value == 1) | (value == 4))
    save_set = useful & (critical_save | special_save) & (focus == chop)
    meaning = play_set | save_set
    new_recv = intersect_or(recv, literal_after[receiver], literal_after[receiver])
    good = new_recv & useful[None, :]
    new_recv = jnp.where(new[:, None] & jnp.any(good, -1, keepdims=True), good, new_recv)
    focused = new_recv[focus] & meaning
    # A Level 1 self-prompt needs an already-clued connector; never use an
    # unclued finesse position. Prefer ordinary interpretations when available.
    predecessor = jnp.maximum(jnp.arange(25) - 1, 0)
    compatible = (new_recv[:, predecessor] & touched[receiver, :, None]
                  & (jnp.arange(5) != focus)[:, None])
    connector = jnp.max(jnp.where(compatible, jnp.arange(5)[:, None], -1), axis=0)
    prompt_set = (new_recv[focus] & useful & (ranks == heights[colors] + 1)
                  & (connector >= 0) & ~other_have[predecessor])
    need_prompt = ~jnp.any(focused) & jnp.any(prompt_set)
    chosen = jnp.max(jnp.where(prompt_set, connector, -1))
    same_connector = jnp.all(~prompt_set | (connector == chosen))
    use_prompt = need_prompt & same_connector
    single_connector = jnp.sum(jnp.any(compatible & prompt_set[None, :], axis=-1)) == 1
    focused = jnp.where(need_prompt, prompt_set, focused)
    new_recv = new_recv.at[focus].set(jnp.where(jnp.any(focused), focused, new_recv[focus]))
    connector_meaning = jnp.concatenate([prompt_set.reshape(5, 5)[:, 1:], jnp.zeros((5, 1), jnp.bool_)], axis=1).reshape(25)
    connector_note = new_recv[jnp.maximum(chosen, 0)] & connector_meaning
    new_recv = new_recv.at[jnp.maximum(chosen, 0)].set(jnp.where(use_prompt, connector_note, new_recv[jnp.maximum(chosen, 0)]))
    # Only the cards present when the clue was given may support this
    # ambiguous ordinary delay. Later draws cannot retroactively justify it.
    ordinary_support = ready_cards[:, predecessor].T
    conditional = (play_set & ~save_set & new_recv[focus] & (ranks == heights[colors] + 1)
                   & ~other_have[predecessor])
    ordinary_support = ordinary_support & conditional[:, None]
    return notes.at[receiver].set(new_recv), focus, jnp.any(save_set & new_recv[focus]), jnp.where(use_prompt, chosen, -1).astype(jnp.int32), single_connector, ordinary_support


def beliefs(notes, counts, visible, touched):
    available = jnp.maximum(counts - visible.sum(0), 0)
    own = notes[0] * available[None, :]
    own = jnp.where(own.sum(-1, keepdims=True) > 0, own, notes[0].astype(jnp.float32))
    own /= jnp.maximum(own.sum(-1, keepdims=True), 1)
    partner = notes[1].astype(jnp.float32) * counts[None, :]
    partner /= jnp.maximum(partner.sum(-1, keepdims=True), 1)
    return jnp.stack([own, partner])



def exact_own_belief(mask, available, occupied, fallback):
    """Exact five-slot marginals, using subset DP and physical multiplicities.

    Each type can fill any compatible subset of the five labeled slots. The
    falling factorial counts the distinct physical copies assigned to it.
```

```
    Differentiating the partition sum gives all 125 marginal numerators.
    """
    subsets = jnp.arange(32, dtype=jnp.int32)
    bits = (subsets[:, None] & (1 << jnp.arange(5))[None, :]) != 0
    sizes = bits.sum(-1)
    allowed = (subsets[None, :] & subsets[:, None]) == subsets[None, :]
    previous = subsets[:, None] ^ subsets[None, :]
    copies = jnp.concatenate([available, jnp.array([5.0])])
    weights = jnp.concatenate([mask.astype(jnp.float32), (~occupied)[:, None].astype(jnp.float32)
    ↪ ], axis=1)
    falling = jnp.prod(jnp.where(jnp.arange(5)[None, None, :] < sizes[None, :, None],
                                 jnp.maximum(copies[:, None, None] - jnp.arange(5)[None, None, :],
    ↪ 0), 1), axis=-1)
    def partition(w):
        def one_type(dp, inputs):
            column, multiplicity = inputs
            term = jnp.prod(jnp.where(bits, column[None, :], 1.0), axis=-1) * multiplicity
            dp = jnp.sum(jnp.where(allowed, dp[previous] * term[None, :], 0.0), axis=-1)
            return dp, None
        initial = jnp.zeros(32).at[0].set(1.0)
        final, _ = jax.lax.scan(one_type, initial, (w.T, falling))
        return final[31]
    normalizer, gradient = jax.value_and_grad(partition)(weights)
    marginals = (gradient * weights / jnp.maximum(normalizer, 1e-20))[:, :25]
    marginals /= jnp.maximum(marginals.sum(-1, keepdims=True), 1e-20)
    return jnp.where(normalizer > 0, marginals, fallback), normalizer > 0

def agent_act(obs, reset, memory, rng):
    m = jax.tree.map(lambda a, b: jnp.where(reset, a, b), init_agent(), memory)
    f = obs.features
    hand, literal, touched, occupied, heights, counts, useful, playable = decode(f)
    _, prev_lit, prev_touched, prev_occ, _, _, _, _ = decode(m['previous'])
    last = f[253:308]
    actor = jnp.argmax(last[:2])
    is_remove = (last[2] + last[3]) > 0
    is_clue = (last[4] + last[5]) > 0
    removed = jnp.argmax(last[23:28])
    notes = m['notes']
    dependencies = m['dependency']
    exempt = m['exempt']
    wait = m['wait']
    was_waiting = jnp.any(wait, axis=-1)
    revealed = jnp.argmax(last[28:53])
    waiting = dependencies[actor] == removed
    linked_meaning = (jnp.repeat(jnp.arange(5), 5) == revealed // 5) & (jnp.tile(jnp.arange(5), 5)
    ↪  == revealed % 5 + 1)
    narrowed_link = notes[actor] & linked_meaning[None, :]
    resolve_link = waiting & is_remove & (last[2] > 0) & (last[53] > 0) & jnp.any(narrowed_link,
    ↪ -1)
    notes = notes.at[actor].set(jnp.where(resolve_link[:, None], narrowed_link, notes[actor]))
    shifted_dep = jnp.where(dependencies[actor] == removed, -1, dependencies[actor] - (
    ↪ dependencies[actor] > removed))
    shift_idx = jnp.minimum(jnp.arange(5) + (jnp.arange(5) >= removed), 4)
    shifted = notes[actor, shift_idx]
    shifted = shifted.at[4].set(jnp.ones(25, jnp.bool_))
    notes = notes.at[actor].set(jnp.where(is_remove & ~reset, shifted, notes[actor]))
    shifted_dep = shifted_dep[shift_idx].at[4].set(-1)
    dependencies = dependencies.at[actor].set(jnp.where(is_remove & ~reset, shifted_dep,
    ↪ dependencies[actor]))
    shifted_exempt = exempt[actor, shift_idx].at[4].set(False)
    exempt = exempt.at[actor].set(jnp.where(is_remove & ~reset, shifted_exempt, exempt[actor]))
    shifted_wait = wait[actor][:, :, shift_idx].at[:, :, 4].set(False)
    shifted_wait = shifted_wait[shift_idx].at[4].set(False)
    shifted_was_waiting = was_waiting[actor, shift_idx].at[4].set(False)
    wait = wait.at[actor].set(jnp.where(is_remove & ~reset, shifted_wait, wait[actor]))
    was_waiting = was_waiting.at[actor].set(jnp.where(is_remove & ~reset, shifted_was_waiting,
    ↪ was_waiting[actor]))
    early = m['early'] & ~(is_remove & (last[3] > 0) & (removed == chop_of(prev_touched[actor],
    ↪ prev_occ[actor])))
    receiver = jnp.argmax(last[6:8])
    other_cards = jnp.where(receiver == 0, hand > 0, notes[0] & (notes[0].sum(-1) == 1)[:, None])
    interpreted, last_focus, last_save, last_connector, last_linkable, new_wait = clue_notes(notes
    ↪ , literal, prev_touched, prev_occ, receiver, last[18:23] > 0, last[5] > 0, jnp.argmax(last
    ↪ [13:18]), heights, counts, useful, other_cards)
    notes = jnp.where(is_clue & ~reset, interpreted, notes)
    wait = wait.at[receiver, last_focus].set(jnp.where(is_clue & ~reset, new_wait, wait[receiver,
    ↪ last_focus]))
    was_waiting = was_waiting.at[receiver, last_focus].set(jnp.where(is_clue & ~reset, jnp.any(
    ↪ new_wait, -1), was_waiting[receiver, last_focus]))
    dependencies = dependencies.at[receiver, last_focus].set(jnp.where(is_clue & ~reset, jnp.where
    ↪ (last_linkable, last_connector, -1), dependencies[receiver, last_focus]))
    rank_two_save = is_clue & (last[5] > 0) & (last[14] > 0) & (last_focus == chop_of(prev_touched
    ↪ [receiver], prev_occ[receiver]))
    other_possible = jnp.where(receiver == 0, hand > 0, notes[0])
    clued_other_possible = jnp.any(other_possible & prev_touched[1-receiver, :, None], axis=0)
```

```
    chop_exception = is_clue & (last_focus == chop_of(prev_touched[receiver], prev_occ[receiver]))
    exceptional_types = playable | (rank_two_save & (jnp.tile(jnp.arange(5), 5) == 1))
    save_exempt = clued_other_possible & exceptional_types & literal[receiver, last_focus] & useful
    # A legitimate save can also duplicate a previously saved hidden card in
    # the giver's own hand; preserve that reciprocal alternative.
    target_types = jnp.where(receiver == 1, hand[last_focus] > 0, interpreted[receiver, last_focus]) & useful & exceptional_types
    reciprocal = prev_touched[1-receiver, :, None] & literal[1-receiver] & target_types[None, :]
    exempt = exempt.at[1-receiver].set(exempt[1-receiver] | (chop_exception & reciprocal))
    old_exempt = exempt[receiver, last_focus]
    exempt = exempt.at[receiver, last_focus].set(jnp.where(chop_exception, old_exempt | save_exempt, jnp.where(is_clue, jnp.zeros(25, jnp.bool_), old_exempt)))
    notes = tidy(notes, literal, touched, useful, hand, counts, heights, exempt)
    predecessor = jnp.maximum(jnp.arange(25) - 1, 0)
    still_possible = jnp.transpose(notes[:, :, predecessor], (0, 2, 1))
    wait = wait & still_possible[:, None, :, :]
    supported_by_board = jnp.tile(jnp.arange(5), 5) <= jnp.repeat(heights, 5)
    broken_delay = was_waiting & ~jnp.any(wait, -1) & ~supported_by_board[None, None, :]
    revised = notes & ~broken_delay
    notes = jnp.where(jnp.any(revised, -1, keepdims=True), revised, literal)
    wait = wait & ~supported_by_board[None, None, :, None]
    # A model of the partner's belief is uncertain; never treat a demonstrably
    # wrong predicted identity as public card evidence.
    partner_consistent = jnp.any(notes[1] & (hand > 0), axis=-1)
    notes = notes.at[1].set(jnp.where(partner_consistent[:, None], notes[1], literal[1]))
    wait = wait.at[1].set(wait[1] & partner_consistent[:, None, None])
    probs = beliefs(notes, counts, hand, touched)
    exact_probs, count_consistent = exact_own_belief(notes[0], jnp.maximum(counts - hand.sum(0), 0), occupied[0], probs[0])
    literal_fallback = beliefs(notes.at[0].set(literal[0]), counts, hand, touched)[0]
    exact_probs = jnp.where(count_consistent, exact_probs, literal_fallback)
    notes = notes.at[0].set(jnp.where(count_consistent, notes[0], literal[0]))
    dependencies = dependencies.at[0].set(jnp.where(count_consistent, dependencies[0], -1))
    wait = wait.at[0].set(wait[0] & count_consistent)
    probs = probs.at[0].set(exact_probs)
    pplay = (probs * playable).sum(-1)
    ptrash = (probs * ~useful).sum(-1)
    ranks = jnp.tile(jnp.arange(5), 5)
    colors = jnp.repeat(jnp.arange(5), 5)
    tokens = f[192:200].sum()
    lives = f[200:203].sum()
    deck = f[127:167].sum()
    partner_id = jnp.argmax(hand, -1)
    partner_live = occupied[1] & useful[partner_id]
    pchop = chop_of(touched[1], occupied[1])
    chop_id = partner_id[jnp.minimum(pchop, 4)]
    partner_ready = jnp.any((pplay[1] > .999) & occupied[1])
    partner_trash = jnp.any((ptrash[1] > .999) & occupied[1])
    unique_chop = jnp.sum((partner_id == chop_id) & occupied[1]) == 1
    chop_danger = unique_chop & (pchop < 5) & useful[chop_id] & ((counts[chop_id] == 1) | (ranks[chop_id] == 1) | playable[chop_id])
    urgent = chop_danger & ~partner_ready & ~partner_trash
    next_chop = chop_of(touched[1] | (jnp.arange(5) == pchop), occupied[1])
    next_id = partner_id[jnp.minimum(next_chop, 4)]
    next_danger = (next_chop < 5) & useful[next_id] & ((counts[next_id] == 1) | (ranks[next_id] == 1))
    proactive_save = chop_danger & next_danger & ~partner_trash & (jnp.sum((pplay[1] > .999) & occupied[1]) <= 1)
    own_certain = jnp.any((probs[0] > .99999) & touched[0, :, None], axis=0)
    own_claim = jnp.any((probs[0] > .00001) & touched[0, :, None], axis=0)
    known_duplicate = own_claim[partner_id]
    same_card = partner_id[:, None] == partner_id[None, :]
    forced = (tokens == 8) | jnp.all(touched[0] | ~occupied[0])

    def evaluate_hint(a):
        rank_hint = a >= 5
        val = a % 5
        touch = occupied[1] & jnp.where(rank_hint, ranks[partner_id] == val, colors[partner_id] == val)
        focus, chop, new = focus_of(touch, touched[1], occupied[1])
        fid = partner_id[focus]
        # Literal positive and negative information supplied by the clue.
        attribute = jnp.where(rank_hint, ranks == val, colors == val)
        after = literal.at[1].set(literal[1] & (touch[:, None] == attribute[None, :]))
        predicted, _, _, prompt_connector, prompt_single, _ = clue_notes(notes, after, touched, occupied, 1, touch, rank_hint, val, heights, counts, useful, notes[0] & (notes[0].sum(-1) == 1)[:, None])
        tagged = touched.at[1].set(touched[1] | touch)
        save_two = rank_hint & (val == 1) & (focus == chop)
        exceptional_types = playable | (save_two & (ranks == 1))
        pred_exempt = exempt.at[1, focus].set(jnp.where(focus == chop, exempt[1, focus] | (own_claim & after[1, focus] & useful & exceptional_types), jnp.zeros(25, jnp.bool_)))
        pred_exempt = pred_exempt.at[0].set(pred_exempt[0] | ((focus == chop) & touched[0, :, None] & literal[0] & (hand[focus] > 0)[None, :] & exceptional_types[None, :]))
```

```
        predicted = tidy(predicted, after, tagged, useful, hand, counts, heights, pred_exempt)
        pr = beliefs(predicted, counts, hand, tagged)[1]
        next_play = (pr * playable).sum(-1) > .999
        newly_ready = jnp.sum(next_play & ~(pplay[1] > .999) & occupied[1])
        next_trash = (pr * ~useful).sum(-1) > .999
        newly_trash = jnp.sum(next_trash & ~(ptrash[1] > .999) & occupied[1])
        literal_gain = jnp.sum(jnp.log1p(literal[1].sum(-1)) - jnp.log1p(after[1].sum(-1)))
        # Only established connectors authorize delayed clues given by this policy.
        have = own_certain | jnp.any((hand > 0) & touched[1, :, None], axis=0)
        accounted = have.reshape(5, 5) | (jnp.arange(5)[None, :] < heights[:, None])
        connected = jnp.concatenate([jnp.ones((5, 1), jnp.bool_), jnp.cumprod(accounted[:, :4],
    ↪ axis=1).astype(jnp.bool_)], axis=1).reshape(25)
        play_ok = (connected[fid] | (prompt_connector >= 0)) & useful[fid]
        save_ok = (focus == chop) & useful[fid] & ((rank_hint & ((val == 1) | (val == 4))) | ((
    ↪ counts[fid] == 1) & (ranks[fid] != 4)))
        duplicate = jnp.any(same_card & (touch | touched[1])[None, :] & ~jnp.eye(5, dtype=jnp.
    ↪ bool_), axis=1)
        bad = jnp.sum(new & (~partner_live | duplicate | known_duplicate))
        nnew = new.sum()
        truth_consistent = predicted[1, focus, fid]
        new_errors = jnp.sum((~predicted[1, jnp.arange(5), partner_id]) & notes[1, jnp.arange(5),
    ↪ partner_id] & (touch | touched[1]))
        false_plays = jnp.sum(next_play & ~playable[partner_id] & occupied[1])
        required_save = (focus == chop) & chop_danger
        prompt_correct = (prompt_connector < 0) | (prompt_single & (partner_id[jnp.maximum(
    ↪ prompt_connector, 0)] + 1 == fid) & (ranks[fid] > 0))
        valid = prompt_correct & truth_consistent & ((new_errors == 0) | required_save) & (play_ok
    ↪  | save_ok) & ((nnew > 0) | forced) & ((bad == 0) | required_save)
        protects = touch[jnp.minimum(pchop, 4)] & (pchop < 5)
        postpones = (jnp.any(next_play) | jnp.any(next_trash)) & ~partner_ready & ~partner_trash
        safety = urgent & (protects | postpones)
        value = 5.0 + 2.0 * newly_ready + .75 * nnew - 4.0 * bad + .2 * ~rank_hint
        after_chop = chop_of(touched[1] | touch, occupied[1])
        after_id = partner_id[jnp.minimum(after_chop, 4)]
        exposed = (after_chop < 5) & useful[after_id] & (counts[after_id] == 1) & ~jnp.any(
    ↪ next_play) & ~partner_trash
        value += 1000.0 * safety + 18.0 * (proactive_save & protects) + 1.0 * (save_ok & (counts[
    ↪ fid] == 1))
        value -= 6.0 * exposed + 4.0 * new_errors + 8.0 * false_plays
        value -= 2.5 * partner_ready + 1.2 * (nnew == 0)
        value -= .7 * (tokens <= 2) + .6 * (tokens <= 1)
        fallback = -100.0 - 30.0 * (~prompt_correct) - 30.0 * false_plays - 8.0 * new_errors - 2.0
    ↪  * bad - .2 * nnew + .1 * rank_hint
        # Forced clues may use literal information without inventing a stall
        # signal. Prefer resolving a real play or safe discard to a blind guess.
        forced_information = forced & prompt_correct & truth_consistent & (new_errors == 0) & (
    ↪ false_plays == 0) & (literal_gain > .001)
        information_value = .2 * literal_gain + 4.0 * newly_ready + 3.0 * newly_trash + 1000.0 *
    ↪ safety
        value = jnp.where(valid, value, jnp.where(forced_information, information_value, fallback)
    ↪ )
        valid = valid | forced_information
        # Forced legal clues carry no additional convention signal.
        return value, focus, newly_ready, bad, valid

    hint_values, hint_focus, hint_ready, hint_bad, hint_valid = jax.vmap(evaluate_hint)(jnp.arange
    ↪ (10))
    ochop = chop_of(touched[0], occupied[0])
    trash = (ptrash[0] > .999) & occupied[0]
    discard_value = jnp.where(trash, 1.0 + .02 * (4-jnp.arange(5)), -30.0)
    discard_value = jnp.where(jnp.arange(5) == ochop, .0, discard_value)
    all_clued = ochop == 5
    loss = (probs[0] * useful * (1 + (counts == 1) * (5-ranks))).sum(-1)
    discard_value = jnp.where(all_clued & ~trash, -2.0 - loss, discard_value)
    # A connector play can itself give the partner a safe action and protect chop.
    boards_after = heights[None, :] + jax.nn.one_hot(colors, 5, dtype=jnp.int32) * playable[:,
    ↪ None]
    playable_after = ranks[None, :] == boards_after[:, colors]
    other_after = jnp.einsum('st,it->is', probs[1], playable_after.astype(jnp.float32))
    protected_after = jnp.any((other_after > .999) & occupied[1][None, :], axis=-1)
    play_protects = (probs[0] * protected_after[None, :]).sum(-1) > .999
    newly_ready_after = jnp.sum((other_after > .999) & ~(pplay[1] > .999)[None, :] & occupied[1][[
    ↪ None, :], axis=-1)
    unlock = (probs[0] * newly_ready_after[None, :]).sum(-1)
    play_value = 16.0 + .6 * (tokens < 8) * (probs[0] * (ranks == 4)).sum(-1) + .02 * (4-jnp.
    ↪ arange(5))
    play_value += 1000.0 * urgent * play_protects + 18.0 * proactive_save * (unlock > .999) + 1.5
    ↪ * unlock
    safe = pplay[0] > .999
    threshold = jnp.where(lives >= 3, .985, jnp.where(lives >= 2, .995, .9999))
    risky = (pplay[0] >= threshold) & touched[0]
    play_value = jnp.where(safe, play_value, jnp.where(risky, 2.0 + pplay[0], -10000.0))
    # At the final two turns, unused information has no value beyond enabling a play.
    end_value = pplay[0] - (1-pplay[0]) * jnp.where(lives > 1, 0.0, heights.sum())
```

```
    play_value = jnp.where((deck == 0) & ~safe, jnp.where(end_value > 0, 2*end_value+2, -10000), play_value)
    first_final_turn = (deck == 0) & (m['previous'][127:167].sum() > 0)
    final_hint = (hint_ready > 0) & hint_valid & ~partner_ready & first_final_turn
    hint_values = jnp.where(deck == 0, jnp.where(final_hint, 4.0, -500.0), hint_values)
    values = jnp.concatenate([discard_value, play_value, hint_values, jnp.array([0.0])])
    action = jnp.argmax(jnp.where(obs.legal_actions, values, -1e6)).astype(jnp.int32)
    out = {'notes': notes, 'previous': f, 'early': early, 'dependency': dependencies, 'exempt': exempt, 'wait': wait}
    trace = {'wait_count': wait.sum((-1, -2)), 'broken_delay': broken_delay.sum(-1), 'count_consistent': count_consistent, 'pplay': pplay, 'ptrash': ptrash, 'notes': notes, 'hint_values': hint_values,
             'hint_focus': hint_focus, 'hint_valid': hint_valid, 'hint_bad': hint_bad,
             'dependency': dependencies, 'last_connector': last_connector, 'last_linkable': last_linkable, 'last_focus': last_focus, 'last_save': last_save, 'urgent': urgent, 'early': early}
    return action, out, trace
```

### G.4 CLAUDE FABLE 5.1 ON OVERCOOKED V2: TEST-TIME (WIDE)

Maintains beliefs over alternative partner-signalling conventions and updates them during play to guide recipe selection and cooking.

```
"""Overcooked V2 test_time_wide policy.

Layout facts (see context/environment/README.md):
  * agent_0 lives in the top room (rows 1-2, cols 1-4) and is the only seat that
    can reach ingredients 0/1, plates, pots (from above) and the delivery station.
  * agent_1 lives in the bottom room (rows 4-5, cols 1-4) and always sees the
    always-on recipe indicator at (6,2); agent_0 never can.
  * Recipes are [0,0,0] or [1,1,1]; ingredient 3 (bottom room) is never needed.

Roles:
  * agent_0: solo cook.  Pot (3,1) <- ingredient 0, pot (3,3) <- ingredient 1,
    keep both pots loaded, fetch plate, choose which dish to deliver from a
    belief over the recipe, deliver, refill.  The recipe belief is a small
    Bayesian filter over hypotheses "partner feature phi (with polarity) encodes
    the recipe" plus "no information".  With an uninformative partner it
    degrades to the info-free alternating strategy (which has positive
    expected value because a wrong delivery does not resample the recipe).
  * agent_1: pure signaller.  Stands in row 4 (visible from agent_0's row 2):
    recipe 0 -> cell (4,1) facing LEFT, recipe 1 -> cell (4,3) facing RIGHT.
    This satisfies column-threshold decoders, "below pot k" decoders and
    left/right facing decoders simultaneously.
"""

import jax
import jax.numpy as jnp

POLICY_API_VERSION = 1
POLICY_METADATA = {'name': 'solo_cook_signal_decoder', 'adapter': 'overcooked_v2', '
    ↪ adapter_version': 1, 'max_height': 7, 'max_width': 6, 'num_ingredients': 4}

# facing ids (observation)
UP, DOWN, RIGHT, LEFT = 0, 1, 2, 3
# action ids
A_RIGHT, A_DOWN, A_LEFT, A_UP, A_STAY, A_INTERACT = 0, 1, 2, 3, 4, 5


def _dir_to_act():
    return jnp.array([A_UP, A_DOWN, A_RIGHT, A_LEFT], dtype=jnp.int32)


# ---- hypothesis space for the recipe-signal decoder (agent_0) ----
# features: 0 col>=3, 1 col>=2, 2 facing LEFT, 3 facing RIGHT, 4 facing UP,
#           5 facing DOWN, 6 item on counter (3,2), 7 partner holding something
# polarity 0: feature==1 <-> recipe 1;  polarity 1: feature==1 <-> recipe 0
NF = 8
NH = 1 + 2 * NF


def _hyp():
    h_feat = jnp.array([-1, 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7], dtype=jnp.int32)
    h_pol = jnp.array([0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1], dtype=jnp.int32)
    h_prior = jnp.array([0.06,
                         0.24, 0.015,
                         0.16, 0.01,
                         0.01, 0.06,
                         0.06, 0.01,
                         0.02, 0.02,
                         0.02, 0.02,
                         0.02, 0.01,
                         0.015, 0.015], dtype=jnp.float32)
    # predicted feature value for recipe r under hypothesis h: r XOR pol
    r_idx = jnp.array([0, 1], dtype=jnp.int32)
    h_pred = (r_idx[None, :] ^ h_pol[:, None]).astype(jnp.int32)  # [NH,2]
    return h_feat, h_pol, h_prior, h_pred


LIKE_OK = 1.8    # likelihood ratio vs. noise when observation matches
LIKE_BAD = 0.2   # when it contradicts
GRACE = 10       # steps after a delivery before the signal is re-anchored


def _i32(x):
    return jnp.asarray(x, dtype=jnp.int32)


def _local(gr, gc, my_r, my_c):
    i = gr - my_r + 2
    j = gc - my_c + 2
    valid = (i >= 0) & (i <= 4) & (j >= 0) & (j <= 4)
```

```
    return jnp.clip(i, 0, 4), jnp.clip(j, 0, 4), valid


def _goto(my_r, my_c, my_dir, tr, tc, tdir):
    """Move to (tr,tc) (horizontal first), then turn to tdir.

    Returns (action_before_arrival_or_turn, arrived_and_facing).
    All turn actions used here are toward blocked cells so they never move.
    """
    dc = tc - my_c
    dr = tr - my_r
    move_h = jnp.where(dc > 0, A_RIGHT, A_LEFT)
    move_v = jnp.where(dr > 0, A_DOWN, A_UP)
    at = (dc == 0) & (dr == 0)
    facing = my_dir == tdir
    act = jnp.where(dc != 0, move_h, jnp.where(dr != 0, move_v, _dir_to_act()[tdir]))
    return _i32(act), at & facing


def _goto_interact(my_r, my_c, my_dir, tr, tc, tdir):
    act, done = _goto(my_r, my_c, my_dir, tr, tc, tdir)
    return jnp.where(done, _i32(A_INTERACT), act)


def _goto_wait(my_r, my_c, my_dir, tr, tc, tdir):
    act, done = _goto(my_r, my_c, my_dir, tr, tc, tdir)
    return jnp.where(done, _i32(A_STAY), act)


# ------------------------------------------------------------------ agent 0

def init_agent_0():
    _, _, h_prior, _ = _hyp()
    return {
        't': _i32(0),
        't_deliv': _i32(0),
        'ndeliv': _i32(0),
        'd_last': _i32(0),
        'W': h_prior[:, None] * jnp.full((NH, 2), 0.5, dtype=jnp.float32),
        'anchor': jnp.full((NF,), -1, dtype=jnp.int32),
        'prev_anchor': jnp.full((NF,), -1, dtype=jnp.int32),
        'spurious': jnp.zeros((NF,), dtype=jnp.int32),
        'nchg': jnp.zeros((NF,), dtype=jnp.int32),
        'prev_cooked_type': _i32(-1),
        'pot0_mem': jnp.array([0, 0, 0, 0, 1], dtype=jnp.int32),  # total, cooking, ready, bad,
    ↪ accepts
    }


def _pot_state(obs, gr, gc, my_r, my_c):
    i, j, valid = _local(gr, gc, my_r, my_c)
    cell = obs[i, j]
    cnt = cell[34:38]
    total = cnt.sum()
    cooked = cell[33] > 0
    cooking = cell[44] > 0
    accepts = (total < 3) & (~cooking) & (~cooked)
    bad = (cnt[2] + cnt[3] > 0) | ((cnt[0] > 0) & (cnt[1] > 0))
    return valid, _i32(total), cooking, cooked, bad, accepts


def _counter_empty(obs, gr, gc, my_r, my_c):
    """True if counter (gr,gc) is visible and empty, or not visible (assume empty)."""
    i, j, valid = _local(gr, gc, my_r, my_c)
    occupied = obs[i, j, 32:38].sum() > 0
    return (~valid) | (~occupied)


def agent_0_act(obs, reset, memory, rng):
    init = init_agent_0()
    mem = jax.tree_util.tree_map(lambda a, b: jnp.where(reset, a, b), init, memory)
    obs = obs.astype(jnp.int32)
    H_FEAT, H_POL, _, H_PRED = _hyp()

    # ---- self ----
    gidx = jnp.argmax(obs[:, :, 23].reshape(-1))
    gi, gj = gidx // 5, gidx % 5
    my_r = _i32(2 - gi)
    my_c = _i32(4 - gj)
    my_dir = _i32(jnp.argmax(obs[2, 2, 1:5]))
    inv = obs[2, 2, 5:11]
    has_plate = inv[0] > 0
    is_cooked = inv[1] > 0
    counts = inv[2:6]
    empty_plate = has_plate & (~is_cooked)
    hold_raw = (~has_plate) & (~is_cooked) & (counts.sum() > 0)
```

```
    raw_type = _i32(jnp.argmax(counts))
    good0 = (counts[0] == 3) & (counts[1] == 0) & (counts[2] == 0) & (counts[3] == 0)
    good1 = (counts[1] == 3) & (counts[0] == 0) & (counts[2] == 0) & (counts[3] == 0)
    dish_type = jnp.where(good0, 0, jnp.where(good1, 1, 2))
    cooked_type = jnp.where(is_cooked, dish_type, -1)
    empty_hands = inv.sum() == 0

    # ---- partner ----
    pm = obs[:, :, 11]
    p_vis = pm.sum() > 0
    pidx = jnp.argmax(pm.reshape(-1))
    pi, pj = pidx // 5, pidx % 5
    p_c = my_c + pj - 2
    p_dir = _i32(jnp.argmax(obs[pi, pj, 12:16]))
    p_hold = obs[pi, pj, 16:22].sum() > 0

    # ---- pots ----
    v0, tot0, cooking0, ready0, bad0, acc0 = _pot_state(obs, 3, 1, my_r, my_c)
    v1, tot1, cooking1, ready1, bad1, acc1 = _pot_state(obs, 3, 3, my_r, my_c)
    pm0 = mem['pot0_mem']
    tot0 = jnp.where(v0, tot0, pm0[0])
    cooking0 = jnp.where(v0, cooking0, pm0[1] > 0)
    ready0 = jnp.where(v0, ready0, pm0[2] > 0)
    bad0 = jnp.where(v0, bad0, pm0[3] > 0)
    acc0 = jnp.where(v0, acc0, pm0[4] > 0)
    pot0_mem = jnp.stack([tot0, cooking0.astype(jnp.int32), ready0.astype(jnp.int32),
                          bad0.astype(jnp.int32), acc0.astype(jnp.int32)]).astype(jnp.int32)

    # ---- counters ----
    c32_i, c32_j, c32_valid = _local(3, 2, my_r, my_c)
    c32_item = obs[c32_i, c32_j, 32:38].sum() > 0

    # ---- features ----
    f_val = jnp.stack([
        (p_c >= 3).astype(jnp.int32),
        (p_c >= 2).astype(jnp.int32),
        (p_dir == LEFT).astype(jnp.int32),
        (p_dir == RIGHT).astype(jnp.int32),
        (p_dir == UP).astype(jnp.int32),
        (p_dir == DOWN).astype(jnp.int32),
        c32_item.astype(jnp.int32),
        p_hold.astype(jnp.int32),
    ])
    f_def = jnp.stack([
        p_vis, p_vis, p_vis, p_vis, p_vis, p_vis,
        c32_valid,
        p_vis,
    ])

    # ---- belief: delivery event (cycle end) ----
    t = mem['t']
    W = mem['W']
    anchor = mem['anchor']
    prev_anchor = mem['prev_anchor']
    spurious = mem['spurious']
    nchg = mem['nchg']
    d_prev = mem['prev_cooked_type']
    delivered = (d_prev >= 0) & (d_prev <= 1) & (~is_cooked)
    W0, W1 = W[:, 0], W[:, 1]
    Wd0 = jnp.stack([W0 * 0.5, W1 + W0 * 0.5], axis=1)
    Wd1 = jnp.stack([W0 + W1 * 0.5, W1 * 0.5], axis=1)
    Wt = jnp.where(delivered, jnp.where(d_prev == 0, Wd0, Wd1), W)
    # a feature that has never changed across a delivery carries no evidence,
    # so only ungated (nchg>=1) features earn the "clean cycle" bonus.
    sp_f = jnp.where(anchor >= 0,
                     jnp.where(spurious > 0, LIKE_BAD, jnp.where(nchg >= 1, LIKE_OK, 1.0)),
                     1.0)
    sp_h = jnp.where(H_FEAT >= 0, sp_f[jnp.clip(H_FEAT, 0, NF - 1)], 1.0)
    Wt = jnp.where(delivered, Wt * sp_h[:, None], Wt)
    W = Wt / jnp.maximum(Wt.sum(), 1e-9)
    prev_anchor = jnp.where(delivered & (anchor >= 0), anchor, prev_anchor)
    anchor = jnp.where(delivered, -1, anchor)
    spurious = jnp.where(delivered, 0, spurious)
    t_deliv = jnp.where(delivered, t, mem['t_deliv'])
    ndeliv = mem['ndeliv'] + delivered.astype(jnp.int32)
    d_last = jnp.where(delivered, d_prev, mem['d_last'])

    # ---- belief: signal observation (anchor / re-anchor) ----
    in_window = (t - t_deliv) >= GRACE
    new_evt = f_def & in_window & (anchor < 0)
    chg_evt = f_def & (anchor >= 0) & (f_val != anchor)
    apply_evt = new_evt | chg_evt
    nchg = nchg + (new_evt & (prev_anchor >= 0) & (f_val != prev_anchor)).astype(jnp.int32)
    spurious = spurious | chg_evt.astype(jnp.int32)
    anchor = jnp.where(apply_evt, f_val, anchor)
    hf = jnp.clip(H_FEAT, 0, NF - 1)
```

```
    evt_h = apply_evt[hf] & (H_FEAT >= 0)
    val_h = f_val[hf]
    factor = jnp.where(evt_h[:, None], jnp.where(val_h[:, None] == H_PRED, LIKE_OK, LIKE_BAD), 1.0)
    # boundary change: the recipe changed right after my delivery d_last, so
    # that delivery was correct; this pins the polarity of feature phi.
    bchg = new_evt & (prev_anchor >= 0) & (f_val != prev_anchor)
    s_star = (prev_anchor ^ jnp.clip(mem['d_last'], 0, 1)) & 1
    kill = bchg[hf] & (H_FEAT >= 0) & (H_POL != s_star[hf])
    factor = jnp.where(kill[:, None], 0.05, factor)
    W2 = W * factor
    # features that never changed across a delivery carry no evidence: keep
    # their hypothesis mass frozen (only the pinned recipe belief updates).
    gated = (nchg[hf] == 0) & (H_FEAT >= 0)
    mass_old = W.sum(1)
    mass_new = jnp.maximum(W2.sum(1), 1e-9)
    W3 = jnp.where(gated[:, None], W2 * (mass_old / mass_new)[:, None], W2)
    W = W3 / jnp.maximum(W3.sum(), 1e-9)

    # ---- decision: which dish is wanted ----
    kk = jnp.maximum(ndeliv, t // 45)
    decay = jnp.where(kk <= 1, 1.0, jnp.where(kk == 2, 0.6, jnp.where(kk == 3, 0.2, 0.05)))
    # once some feature has demonstrably tracked a recipe change, features that
    # never changed are much less likely to be the partner's signal.
    any_ungated = (nchg >= 1).any()
    tau_f = jnp.where(nchg >= 1, 1.0, jnp.where(any_ungated, jnp.minimum(decay, 0.2), decay))
    tau_h = jnp.where(H_FEAT >= 0, tau_f[hf], 1.0)
    p0 = (tau_h * W[:, 0]).sum() / jnp.maximum((tau_h * W.sum(1)).sum(), 1e-9)
    d_choice = jnp.where(p0 >= 0.5, 0, 1)

    # ---- controller ----
    ready = jnp.stack([ready0, ready1])
    cooking = jnp.stack([cooking0, cooking1])
    accepts = jnp.stack([acc0, acc1])
    totals = jnp.stack([tot0, tot1])
    pot_col = jnp.array([1, 3], dtype=jnp.int32)

    # dump target: first free counter among (0,4),(0,1),(3,4),(3,2)
    e04 = _counter_empty(obs, 0, 4, my_r, my_c)
    e01 = _counter_empty(obs, 0, 1, my_r, my_c)
    e34 = _counter_empty(obs, 3, 4, my_r, my_c)
    dump_04 = _goto_interact(my_r, my_c, my_dir, 1, 4, UP)
    dump_01 = _goto_interact(my_r, my_c, my_dir, 1, 1, UP)
    dump_34 = _goto_interact(my_r, my_c, my_dir, 2, 4, DOWN)
    dump_32 = _goto_interact(my_r, my_c, my_dir, 2, 2, DOWN)
    e32 = ~c32_item
    act_deliver = _goto_interact(my_r, my_c, my_dir, 1, 2, UP)
    # if every dump slot is full, deliver the bad dish anyway to free the hands
    act_dump = jnp.where(e04, dump_04, jnp.where(e01, dump_01, jnp.where(e34, dump_34, jnp.where(e32, dump_32, act_deliver))))

    # A: holding cooked dish
    act_A = jnp.where(dish_type <= 1, act_deliver, act_dump)

    # B: holding empty plate
    d = d_choice
    other = 1 - d
    avail = ready | cooking
    bad = jnp.stack([bad0, bad1])
    good_avail = avail & (~bad)
    bad_ready = ready & bad
    uncertain = jnp.abs(p0 - 0.5) < 0.15
    d_eff = jnp.where(good_avail[d], d,
                      jnp.where(good_avail[other] & uncertain, other,
                                jnp.where(bad_ready[d], d, jnp.where(bad_ready[other], other, d))))
    pot_ok = good_avail[d_eff] | bad_ready[d_eff]
    scoop = _goto_interact(my_r, my_c, my_dir, 2, pot_col[d_eff], DOWN)
    wait = _goto_wait(my_r, my_c, my_dir, 2, pot_col[d_eff], DOWN)
    wait_center = _goto_wait(my_r, my_c, my_dir, 2, 2, DOWN)
    act_B = jnp.where(pot_ok, jnp.where(ready[d_eff], scoop, wait),
                      jnp.where(cooking[d], wait, act_dump))

    # C: holding raw ingredient (never feed a contaminated pot)
    good_acc = accepts & (~bad)
    k = jnp.clip(raw_type, 0, 1)
    tgt = jnp.where(good_acc[k] & (raw_type <= 1), k, jnp.where(good_acc[1 - k] & (raw_type <= 1), 1 - k, -1))
    put = _goto_interact(my_r, my_c, my_dir, 2, pot_col[jnp.clip(tgt, 0, 1)], DOWN)
    act_C = jnp.where(tgt >= 0, put, act_dump)

    # D: empty hands
    partial0 = good_acc[0] & (tot0 > 0)
    partial1 = good_acc[1] & (tot1 > 0)
    # partially filled pot first; otherwise the pot of the currently believed recipe
    fill_k = jnp.where(partial0, 0, jnp.where(partial1, 1,
```

```
                        jnp.where(good_acc[d], d, jnp.where(good_acc[other], other, -1))))
    get_ing0 = _goto_interact(my_r, my_c, my_dir, 1, 1, LEFT)
    get_ing1 = _goto_interact(my_r, my_c, my_dir, 2, 4, RIGHT)
    get_plate = _goto_interact(my_r, my_c, my_dir, 1, 3, UP)
    plate_useful = good_avail[d] | (good_avail.any() & uncertain) | bad_ready.any()
    act_D = jnp.where(fill_k == 0, get_ing0, jnp.where(fill_k == 1, get_ing1,
                      jnp.where(plate_useful, get_plate, wait_center)))

    action = jnp.where(is_cooked, act_A,
                       jnp.where(empty_plate, act_B,
                                 jnp.where(hold_raw, act_C, act_D)))
    action = _i32(action)

    new_mem = {
        't': t + 1,
        't_deliv': t_deliv,
        'ndeliv': ndeliv,
        'd_last': _i32(d_last),
        'W': W.astype(jnp.float32),
        'anchor': anchor.astype(jnp.int32),
        'prev_anchor': prev_anchor.astype(jnp.int32),
        'spurious': spurious.astype(jnp.int32),
        'nchg': nchg.astype(jnp.int32),
        'prev_cooked_type': _i32(cooked_type),
        'pot0_mem': pot0_mem,
    }
    trace = {
        'p0': p0.astype(jnp.float32),
        'd_choice': _i32(d_choice),
        'ndeliv': ndeliv,
        'nchg': nchg.astype(jnp.int32),
        'wmass': W.sum(1).astype(jnp.float32),
        'p_vis': p_vis,
        'fill_k': _i32(fill_k),
        'hold': _i32(jnp.where(is_cooked, 3, jnp.where(empty_plate, 2, jnp.where(hold_raw, 1, 0)))
    ↪ ),
        'deliv_d': _i32(jnp.where(delivered, d_prev, -1)),
        'p_c': _i32(jnp.where(p_vis, p_c, -1)),
        'anchor': anchor.astype(jnp.int32),
        'spur': spurious.astype(jnp.int32),
        'my_rc': jnp.stack([my_r, my_c]).astype(jnp.int32),
    }
    return action, new_mem, trace


# ------------------------------------------------------------------ agent 1

def init_agent_1():
    return {'recipe': _i32(0), 't': _i32(0)}


def agent_1_act(obs, reset, memory, rng):
    init = init_agent_1()
    mem = jax.tree_util.tree_map(lambda a, b: jnp.where(reset, a, b), init, memory)
    obs = obs.astype(jnp.int32)

    ridx = jnp.argmax(obs[:, :, 25].reshape(-1))
    ri, rj = ridx // 5, ridx % 5
    r_vis = obs[ri, rj, 25] > 0
    my_r = _i32(8 - ri)
    my_c = _i32(4 - rj)
    my_dir = _i32(jnp.argmax(obs[2, 2, 1:5]))
    rc = obs[ri, rj, 40:44]
    known = r_vis & (rc.sum() > 0)
    recipe = jnp.where(known, jnp.where(rc[1] > rc[0], 1, 0), mem['recipe'])
    recipe = _i32(recipe)

    # recipe 0: stand at (4,1) facing LEFT (turn is blocked by supply at (4,0))
    at0 = (my_r == 4) & (my_c == 1)
    act0 = jnp.where(at0 & (my_dir == LEFT), A_STAY,
                     jnp.where(at0, A_LEFT,
                               jnp.where(my_r == 5, A_UP, A_LEFT)))
    # recipe 1: stand at (4,3) facing RIGHT (arrive from (4,2) moving right)
    at1 = (my_r == 4) & (my_c == 3)
    act1 = jnp.where(at1 & (my_dir == RIGHT), A_STAY,
                     jnp.where(my_r == 5, A_UP,
                               jnp.where(my_c < 3, A_RIGHT, A_LEFT)))
    action = _i32(jnp.where(recipe == 1, act1, act0))

    new_mem = {'recipe': recipe, 't': mem['t'] + 1}
    trace = {'recipe': recipe, 'my_r': my_r, 'my_c': my_c}
    return action, new_mem, trace
```

## H EXAMPLES OF EXPERIMENT-LEDGER ENTRIES

**A03: Robustness to alternative signalling conventions**

**Hypothesis.** The policy can cooperate with alternative signallers and decoders while preserving its self-play return.

**Exploratory test.** Test self-generated partner variants over 32 episodes per pairing. Strict positional decoders fail to interpret the two-marker signal; a count-based signaller also exposes a convention mismatch.

**Revision.** Switch the signal encoding when pot inactivity or incorrect ingredients indicate that the partner cannot interpret it.

**Official evaluation.** Mean self-play return $180 \pm 0$ (SD), over 1,024 episodes.

**Decision.** `follow_up`

**Limitations.** The count-based convention remains incompatible. Partners are self-authored variants; independently generated partners may differ.

**A04: Checking false switches with slow partners**

**Hypothesis.** Half-speed partners do not trigger harmful, unnecessary changes to the signal encoding.

**Exploratory test.** Half-speed partner variants obtain returns of 101.9, 101.9, and 103.8, with no harmful false switches reported.

**Revision.** Cap blind deliveries at two when signals are absent. With a no-op signaller, the mean return is $-3.8$; this safeguard reduces return with a flag-based signaller from 165 to 153.1.

**Official evaluation.** Mean self-play return $180 \pm 0$ (SD), over 1,024 episodes.

**Decision.** `keep` **Parent entry.** A03.

**Limitations.** The count-based convention remains incompatible, and the tests do not establish compatibility with independently generated partners.

Figure 12: **Experiment-ledger examples.** Summaries of two consecutive experiments logged by Claude Fable 5.1 on Overcooked V2's `test_time_simple` layout. The `follow_up` decision indicates a planned further policy iteration, while `keep` indicates a decision to retain the policy.