CodeAssist: Training Your Personalized AI Coding Assistant via Tool-Mediation Games

Day 0 *Research Report*[†]

Gensyn AI Team

CodeAssist is a research agenda and app for training personalized AI assistants via *tool-mediation games*–a variant of assistance games in which an assistant calls on tools to take actions in a shared environment with users. In this report we introduce tool-mediation games and present the first application of them to a coding-based tool-mediation game where an AI assistant "pair programs" with a human within a single file. Crucially, in tool-mediation games, assistants cannot directly influence the shared environment and instead call on tools that execute actions for it. In CodeAssist this means the assistant does not learn to code directly; frozen LLMs serve as delegates the assistant prompts to generate content depending on its chosen action. Instead, the assistant learns *where* in the file to interact with, *when* to intervene, and *what* kind of action (e.g. leave a comment vs write a line code) to take in order to best compliment the human user. By utilizing tool-mediation games CodeAssist keeps assistant training well-defined, consumer hardware-friendly, and directly personalized to users; the assistant is a small model that specializes at adapting to a user's coding style and workflow while wielding specialized tools better suited for solving complex sub-tasks.

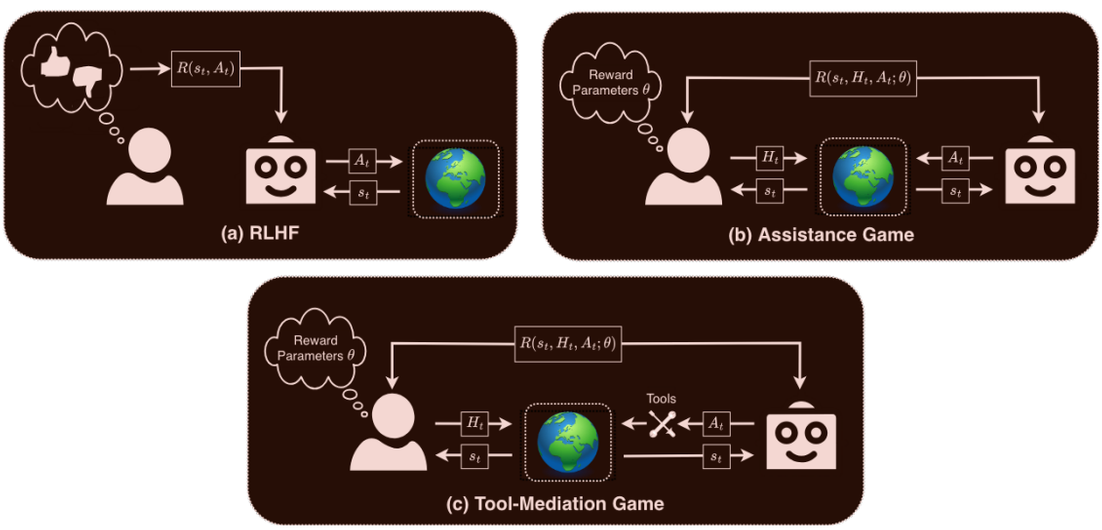

Figure 1 Comparison of assistant training paradigms. RLHF (a) is currently the dominant paradigm where direct human preferences are used for learning and assistants are generally used as exogenous tools by users. In traditional assistance games (b) humans and assistants share a reward function and act in the same environment, whereas in tool-mediation games (c) assistants are restricted to tool-mediated actuation where they select and parametrize a delegate tool (e.g. LLM or API) whose output acts on the shared environment.

[†]With CodeAssist we at Gensyn are taking a unique approach to research; an approach whose transparency reflects the principles we believe must be entrenched in AI development. After the initial release of CodeAssist on 11/05/2025, we will continue to build and iterate on the project completely in the open–the code, the research, and the leanings will all be open source. As part of this commitment we will periodically update this report with results, insights, etc. after each big update made to CodeAssist.

Corresponding author(s): gabriel@gensyn.ai

1. Introduction

The majority of AI assistants today interact with users in essentially single-turn exchanges; they either make small suggestions that a user accepts or rejects, or they attempt to complete a task outright after each individual prompt they receive. The prominence of these constrained interactions is partly due to UI/UX choices and partly a consequence of the standard pipeline for training general AI assistants today–pretraining, then supervised finetuning, and finally reinforcement learning from human feedback (RLHF) or its variants. The final stage of this pipeline, RLHF, aims to align the assistant with human preferences by finetuning pretrained models using direct feedback from human annotators according to criteria such as "helpfulness" and "harmlessness" (Bai et al., 2022; Ouyang et al., 2022). However, *assistance is hard to measure*: feedback must compress a multi-step, collaborative process into scalar labels, and the measured quantity often depends non-linearly on what the assistant decided to do along the way as well as what the human wanted at that specific point in time. As a result, and because existing models are capable of performing many tasks from a single prompt, it is natural for AI assistants trained with RLHF–and their feedback collection–to gravitate towards single-turn exchanges rather than noisier multi-turn, process-level interactions.

Although RLHF works well in environments with these single-turn exchanges where the quality of responses is easy to judge, it has several limitations that are well articulated by Laidlaw et al. (2025): (i) Feedback is subjective and can result in noisy labels for complex tasks that are path-dependent, context-sensitive, and user-specific (Yan et al., 2024; Zhang et al., 2024). (ii) Since these assistants aim to optimize feedback, they are incentivized to over-prioritize what annotators tend to notice (e.g. polish, verbosity) and even utilize deceptive or manipulative behaviours (Lang et al., 2024; Williams et al., 2024; Sharma et al., 2023). (iii) Task-level rewards complicate credit assignment and do not require assistants to maintain uncertainty about a users' goals, which discourages assistants from asking for follow-ups or hedging and can disproportionately assign credit to individual steps while discounting required prerequisites (Shani et al., 2024; Chen et al., 2021). (iv) RLHF treats assistants as exogenous tools that predict users' actions or altogether replace them during tasks (Figure 1a), whereas assistance should be an act of complementing users and cooperating with them (Figure 1b & 1c).

Assistance games sidestep these pitfalls in RLHF by explicitly modeling human-assistant collaboration as cooperative two-player games (Fern et al., 2014; Hadfield-Menell et al., 2016; Laidlaw et al., 2025). The dynamic interactions and uncertainty inherent to assistance are formalized in assistance games (Figure 1b): both agents act in a shared environment while sharing a reward function, but crucially the assistant does not initially know the reward function and must help the human accomplish their goals despite partial observability. This framing aligns assistant objectives with collaborative work; success is defined in the environment instead of post-hoc labels, so assistants are incentivized to work through uncertainty with users and complement them at every step rather than deceive or pander to users.

Despite all of these obvious benefits to assistance games, their lack of widespread adoption is a byproduct of having been generally dismissed as intractable outside of toy problems (Papadimitriou and Tsitsiklis, 1987; Madani et al., 2003) until recent work by Laidlaw et al. (2025) showed they can be practically solved in settings with a large number of actions and goals. In this project our aim is to both extend and generalize the foundations laid by Laidlaw et al. (2025) so that we may show that assistance games can be solved in complex real-world environments.

To this end we introduce *tool-mediation games* (TMGs), a practically useful subclass of assistance games. TMGs are designed for real assistance settings where there may be a competitive ever-evolving set of domain-specific tools (e.g. agents, LLMs, and APIs), and they are defined to scale and adapt with the state of the art by utilizing abstract action sets to call these tools appropriately. The assistant in a TMG *cannot act on the environment directly* and instead, at each turn, utilizes a *delegate tool* (e.g. an LLM) whose output affects the shared state. Formally, as discussed in §3, TMGs are a special case of assistance

games and inherit their properties, but they carve out a regime that matches today's ecosystem of AI products: an abundance of unique, reliable tools that can be managed locally by a small decider trained to assist. By separating decision making from action taking, this split delivers (i) *modularity*: let tools develop and improve on their specialty, and simply retrain the decision policy when it's time to update; (ii) *auditability & safety*: all state changes come from known tools and can be tied directly to a source[1]; (iii) *personalization*: the decision maker learns when/where/how to intervene for a specific user; (iv) *hardware adaptability*: the trainable component can be made small enough to fit on consumer hardware and tools can be chosen to match specific users' needs, constraints, and preferences.

As a first proof of concept of TMGs' practical applicability, we use them to train assistants on a coding task in this project. *CodeAssist* gives users a pair-programming assistant that operates within a single file using policies trained to solve a TMG derived for this task. The assistant's policy is small and learns to decide *when*, *where* (which line/block), and *how* (edit/insert/explain/NO-OP) to act. This design keeps the assistant's learning task well-defined and hardware-friendly while directly optimizing for *helpfulness in the environment* rather than for proxy preferences.

Contents & Statement of Purpose In this research report we will introduce relevant background on assistance games (§2), formalize TMGs (§3), and derive the coding assistance TMG underlying CodeAssist (§3.1). The purpose of this report is to serve as a "living paper" that is periodically updated with results, insights, etc. after each big update made to CodeAssist; all the code is open source on GitHub[2], all research progress will be made public as we update this report, and everyone is encouraged to participate as well as contribute towards the project. Details about the CodeAssist research roadmap and related discussions will be updated in §4. As they become available, results (both positive and negative) will be added to §5. Since this report will likely grow beyond the scope of what is typically in a single paper throughout the duration of the project, we will keep an updated summary of key contributions and broad overview of next steps in the GitHub repo where this report is hosted.

It is our hope that maintaining this report up to date provides users with a rigorous source for understanding what they are contributing towards, and how they can best help drive progress. At key checkpoints in CodeAssist's research roadmap, if we have accumulated sufficient progress and evidence, we will condense the results reported here into an academic paper. By building together it is our belief that we can accelerate research progress while simultaneously building transparent and safe AI systems proven to work in real-world settings.

Summary of Contributions

- We formalize *tool-mediation games* (TMGs) and situate them as a special case of assistance games.
- We present a pair-programming TMG with editor-native states and actions.
- We release *CodeAssist*, a working app that instantiates the pair-programming TMG.

Priorities For Next Update

- Upon release on 11/05/2025, we will begin analyzing preliminary metrics on assistant performance with the baseline policy architecture and supervised PPO-only training.
- Gather user feedback, fix any bugs which may affect metrics in unintended ways, and stabilize the UI/UX needed to embark on roadmap highlighted in §4.1.

[1] Although the tools themselves can have auditability and safety concerns, TMGs give users the ability to vet tools so their assistant only calls on tools the user is comfortable with. In addition, since all state changes are tied to a specific tool, users can easily pinpoint which tools gave questionable or concerning outputs.

[2] https://github.com/gensyn-ai/codeassist

2. Background

Assistance games, first introduced by Fern et al. (2014) and Hadfield-Menell et al. (2016), are two-player Markov games (Littman, 1994; Shapley, 1953) in which a human H and assistant A interact in a shared state to optimize a shared reward function. They are defined by a 6-tuple $(\mathcal{S}, \mathcal{A}^H, \mathcal{A}^A, \Theta, \mathcal{P}, R)$ where $\mathcal{S}$ is the state space, $\mathcal{A}^\alpha$ is the action space for the agent $\alpha \in \{H, A\}$, Θ is a set of possible reward parameters, $\mathcal{P} : \mathcal{S} \times \mathcal{A}^H \times \mathcal{A}^A \to \Delta(\mathcal{S})$ is a state transition distribution, and $R : \mathcal{S} \times \mathcal{A}^H \times \mathcal{A}^A \times \Theta \to \mathbb{R}$ is a reward function. For the purposes of this report, we can assume the state space $\mathcal{S}$ and action spaces $\mathcal{A}^H$, $\mathcal{A}^A$ are finite and discrete; the reward parameters Θ can be any information that encodes the shared goal (e.g. a set of unit tests in coding tasks).

After sampling the initial state $s_0 \in \mathcal{S}$ and reward parameters $\theta \in \Theta$ from a predefined distribution, assistance games unfold as a sequence on discrete timesteps. At each timestep $t \in [T]$, for some $T > 0$, each agent selects their actions $a_t^H \in \mathcal{A}^H$ and $a_t^A \in \mathcal{A}^A$. The agents receive shared reward $R(s_t, a_t^H, a_t^A ; \theta)$ and the environment transitions to the next state by sampling $\mathcal{P}(s_{t+1} \mid s_t, a_t^H, a_t^A)$.

Let $h_t = \{s_0, a_0^H, a_0^A, \ldots, s_{t-1}, a_{t-1}^H, a_{t-1}^A, s_t\}$ be the state-action history up to the current timestep, the policy $\pi^H : (\mathcal{S} \times \mathcal{A}^H \times \mathcal{A}^A)^* \times \mathcal{S} \times \Theta \to \Delta(\mathcal{A}^H)$ define a distribution $\pi^H(a^H \mid h_t, \theta)$ over human actions[3], and the policy $\pi^A : (\mathcal{S} \times \mathcal{A}^H \times \mathcal{A}^A)^* \times \mathcal{S} \to \Delta(\mathcal{A}^A)$ define a distribution over assistant actions $\pi^A(a_t^A \mid h_t)$. Importantly, since the assistant A does not observe the reward parameters, the assistant policy π^A is *not* conditioned on them and instead is only conditioned on h_t which it can observe. In their work introducing assistance games (under the name "cooperative inverse reinforcement learning"), Hadfield-Menell et al. (2016) gave a reduction from optimal joint policy computation in assistance games to solving a single-agent partially observable Markov decision process (POMDP). The elegance of this reduction is that it conveniently allows us to frame solving assistance games as the relatively intuitive task of learning a best response strategy against a fixed human policy. As the remainder of this report will not require a more formal treatment of these ideas, we refer interested readers to the original paper (Hadfield-Menell et al., 2016).

3. Tool-Mediation Games

In this project we utilize a special class of assistance games that we call *tool-mediation games* (TMGs). In TMGs the assistant cannot directly act on the environment; it selects and parametrizes a delegate tool (e.g. LLM or API) whose outputs are used to affect the state, and the assistant's goal is to learn when/where/how to best invoke a tool in response to actions taken by users. Stated simply, in a TMG the *assistants decide actions* while their *tools take actions*.

To formalize this as an assistance game we'll briefly introduce some notation and show how it maps onto a 6-tuple defining an assistance game as introduced in §2. Let $\mathcal{D}$ be a set of delegate tools, $\mathcal{X}$ be a set of parametrizations (e.g. prompts or arguments) needed by those tools, and $\Phi : \mathcal{S} \times \mathcal{D} \times \mathcal{X} \to \mathcal{A}^A$ be an actuation map that transforms decisions made by assistants $(d, x) \in \mathcal{D} \times \mathcal{X}$ into concrete actions taken on the environment. A *tool-mediation assistance game* (TMG) is an assistance game defined by $(\mathcal{S}, \mathcal{A}^H, \{\mathcal{D}, \mathcal{X}, \Phi^A\}, \Theta, \mathcal{P}, R)$ where at each timestep $t \in [T]$ the human H behaves identically to before and the assistant A decides $(d_t, x_t) \in \mathcal{D} \times \mathcal{X}$ which corresponds to an action $a_t^A = \Phi^A(s_t, d_t, x_t) \in \mathcal{A}^A$; they receive shared reward $R(s_t, a_t^H, \Phi^A(s_t, d_t, x_t) ; \theta)$; and the environment transitions to s_{t+1} according to the distribution $\mathcal{P}(s_{t+1} \mid s_t, a_t^H, \Phi^A(s_t, d_t, x_t))$.

[3] Hadfield-Menell et al. (2016) showed that conditioning policies on h_t is not required for finding an optimal human policy. We include it for completeness as we utilize it in notation for CodeAssist's TMG.

3.1. The CodeAssist Tool-Mediation Game

Underlying CodeAssist, and the primary TMG we will focus on throughout this project, is a specific TMG derived for training pair-programming assistants. In this section we will carefully introduce each component of this TMG as it will both serve as the basis for CodeAssist and as a blueprint for deriving TMGs in other assistance settings.

The state space $\mathcal{S}$. The environment in CodeAssist is a file with $L > 0$ lines that are each at most $W > 0$ characters wide. We can represent the file as an ordered sequence $\mathcal{F} = \{\ell_i \in \Sigma^{\leq W}\}_{i \in [L]}$, where Σ is some alphabet and $\Sigma^{\leq W}$ is the set of all strings over Σ of length less than or equal to W. The state space $\mathcal{S}$ contains all possible configurations of file $\mathcal{F}$, and each state $s \in \mathcal{S}$ is a particular set of lines composed of character from Σ under the constraints imposed by L and W.

The action sets $\mathcal{A}^{\alpha \in \{H,A\}}$. Actions on $\mathcal{F}$ in CodeAssist equate to any legal edit(s) to $\mathcal{F}$, which becomes intractably large even with reasonable choices for L, W, and Σ. For example, in a relatively small file having 100 lines ($L = 100$) with at most 100 characters each ($W = 100$) and an alphabet of ASCII printable characters ($|\Sigma| = 128$), just the subset of actions equating to single character replacements has size 1.28×10^6 which is orders of magnitude larger than MBAG's entire action set (Laidlaw et al., 2025). When training assistant and human models in CodeAssist we dramatically decrease the complexity of $\mathcal{A}^{\alpha \in \{H,A\}}$ by first abstracting into a coarse-grained set of decisions and then mapping these decisions to concrete actions using an actuation map. Specifically, for agent $\alpha \in \{H, A\}$, all actions $a^\alpha \in \mathcal{A}^\alpha$ in CodeAssist can be abstracted into one of the coarse-grained decisions described in Definition 1.

Definition 1. The CodeAssist decision set `Decisions` contains the following elements:

1. `NO-OP`, i.e. make no edits to the $\mathcal{F}$.
2. `Fill Partial Line`, i.e. complete $\ell_i \in \mathcal{F}$ where i corresponds to the line the cursor is on.
3. `Write Single Line`, i.e. replace $\ell_i \in \mathcal{F}$ where i corresponds to the line the cursor is on.
4. `Write Multi Line`, i.e. replace $\ell_i \in \mathcal{F}$ and append new lines after it where i corresponds to the line the cursor is on.
5. `Edit Existing Lines`, i.e. replace any set of individual $\ell_j \in \mathcal{F}$ without cursor constraints.
6. `Explain Single Lines`, i.e. append comments to any set of $\ell_j \in \mathcal{F}$ without cursor constraints.
7. `Explain Multi Lines`, i.e. prepend comments to any set of $\ell_j \in \mathcal{F}$ without cursor constraints.

Given a decision in `Decisions`, we make the simplifying assumption that there exists an actuation map for each agent that translates them into actual comments, lines of code, etc. applied on $\mathcal{F}$. For assistant and human models we define (§4) the actuation map as a call to delegate tools, where each of these decisions corresponds to specific tool calls. Actual humans can also be modeled using this abstraction by defining an actuation map from the state-action history h_t and their decisions onto $\mathcal{A}^H$, we discuss this briefly at the end of §4.1.1.

The reward parameters Θ and reward function R. In the first iteration of CodeAssist each coding goal is defined by a set of test cases that need to be passed. Hence the reward parameters Θ consist of this set of tests as well as "line survival" and `NO-OP` related parameters; further details about these reward signals are given in §4.1.2. The reward function R takes these reward parameters and state-action tuples at each timestep then maps them to a reward signal.

The state transition distribution $\mathcal{P}$. Given a state $s \in \mathcal{S}$ and an action $a^\alpha \in \mathcal{A}^\alpha$ for $\alpha \in \{H, A\}$, the action a^α can be thought of as acting on s to yield some $s' \in \mathcal{S}$. A convenient way to represent this for

coding tasks, and how we implemented it in CodeAssist, is to let $a^\alpha \in \mathcal{A}^\alpha$ be represented as unified diffs we can apply to the file. Borrowing notation from algebraic group actions, we use $s \cdot a^\alpha$ to denote agent α's action acting on s. Therefore, at any given timestep $t \in [T]$ we can succinctly define a state-action trajectory by $s_{t+1} = s_t \cdot a_t^H \cdot a_t^A = s_0 \cdot a_0^H \cdot a_0^A \ldots a_{t-1}^H \cdot a_{t-1}^A \cdot a_t^H \cdot a_t^A$. Building on this observation and following the design of AssistanceZero (Laidlaw et al., 2025), we will simplify our implementation and analysis in CodeAssist by assuming the human agent H takes an action followed by the assistant agent A taking an action at every timestep.

4. CodeAssist Research Roadmap

We organize CodeAssist's research roadmap into three phases:

- In Phase 1 (Figure 2) we will focus on LeetCode-style coding tasks as we measure performance metrics, address user feedback, and work to gradually build out an end-to-end training pipeline in the spirit of AssistanceZero (Laidlaw et al., 2025).
- In Phase 2 our primary aim is to move beyond LeetCode-style tasks and towards truly open-ended coding tasks in a single file.
- In Phase 3 our target is generalizing to essentially arbitrary, multi-file coding tasks.

These phases are primarily charted for organizational purposes in this report, and are likely to change as we make progress; hence all content should be seen as a reflection of the current state of CodeAssist and subject to heavy revisions as the project evolves.

In the subsections below we detail the key design decisions and, when available, results corresponding to that phase of the roadmap. Until they become relevant, subsections §4.2 and §4.3 will remain empty but are included in the document to allow for convenient forward references as this report grows.

Current Status: On 11/05/2025 we launched the first version of CodeAssist. The current implementation includes many of the core features in terms of UI/UX, but we have purposely constrained the training pipeline to only include a baseline PPO training pass on human-recorded episodes. Our most pressing goal is to measure baseline performance with this constrained training pipeline and understand how users interact (or want to interact) with their assistants. Once we have stabilized CodeAssist by incorporating user feedback, we will begin expanding on the training pipeline and turn our gaze towards carefully measuring the impact of additions being made.

4.1. Phase 1: Assistant Training Pipelines

CodeAssist Phase 1 is entirely centered around LeetCode-style coding tasks as we work towards an end-to-end training pipeline in the spirit of AssistanceZero (Laidlaw et al., 2025)–summarized in Figure 2. The primary purpose of this phase is to formalize and iterate on the end-to-end pipeline so that in later phases we can focus on what said pipeline is being applied to.

Our discussion of this phase of the roadmap is organized as follows:

- In §4.1.1 we define how raw state-action trajectories are transformed into an intermediary embedding representation before ultimately being mapped to tool calls made by assistant policies.
- In §4.1.2 we discuss the reward functions and their parameters that are being used in CodeAssist.
- In §4.1.3 we introduce the featurizer and policy model architectures implemented in CodeAssist.
- In §4.1.4 we cover the existing training pipeline, what is needed, and how we will extend it.

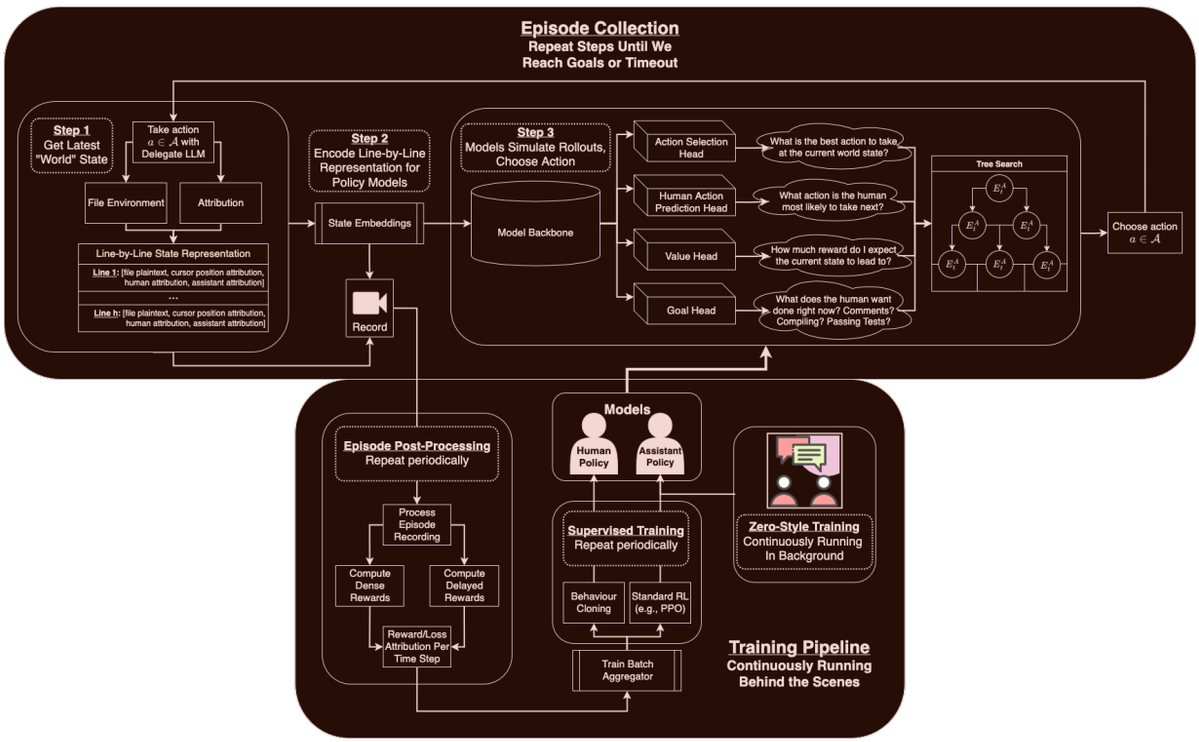

Figure 2 End-to-end execution diagram of CodeAssist that we are building towards in the Phase 1 of the project's development. In the current version of the CodeAssist we have not included zero-style training, any form of tree search, or wired in the goal head. We first want to measure how assistants perform without these components, and will gradually introduce them in order to understand their effects on assistants as well as how they might best be leveraged.

4.1.1. States, Decisions, & Actions

State-action trajectories can be succinctly represented as a sequence of embeddings that agent policies can ingest. There are several ways of representing state-action tuples and mapping them to embeddings, but due to the TMG formulation of CodeAssist we can utilize a granular representation intended to supplement the agents' tool-call decision making. Importantly, although we will describe the mapping from state-action tuples to embeddings as a two step process here, in practice we directly store state-action tuples in the line-by-line representations we introduce next.

State-Action Tuples → Attributed Line-by-Line State Representation Upon being executed, every action $a^\alpha \in \mathcal{A}^\alpha$ can be equivalently described as a set of operations on individual lines $\ell \in \mathcal{F}$. Furthermore, at the granularity of delegated tool-calls in $\mathcal{D} \times \mathcal{X}$ corresponding to elements of `Decisions`, each such operation taken on a line can be fully described by three properties: (i) Which agent executed said operation (e.g. H vs A). (ii) Whether the operation was to write, delete, or both. (iii) Which character indices on the line were operated on.

Recall that, by the definition of $\mathcal{F}$ in §3.1, file states can be represented as an ordered set of lines. Putting these observations together, at any time step $t \in [T]$, we can fully capture state-action tuples in terms of the decisions in `Decisions` using a line-by-line representation where each line $\ell \in \mathcal{F}$ has the following details corresponding to it: the actual text on line ℓ, information about the cursor's position in the file relative to ℓ (is the cursor on ℓ, on which character, etc.), and operations (if any) taken by H and/or A on line ℓ during this timestep.

In fact, as implemented in CodeAssist, we enrich these line-by-line representations without loss of

generality by instead utilizing the following more general representation.

Definition 2. The *attributed line-by-line state representation* is an ordered set such that, for each line $\ell \in \mathcal{F}$ we keep the following details:

1. The actual text on ℓ,
2. Cursor information relative to ℓ (e.g. when it was last there and which character it was on),
3. The most recent operations (if any) taken by H on ℓ and what timestep it was taken on.
4. The most recent operations (if any) taken by A on ℓ and what timestep it was taken on.

Importantly, before concluding this discussion, we note three benefits to using this attributed line-by-line state representation. First, the representation captures all information contained in state-decision tuples and more. Secondly, we effectively "break up" individual decisions taken by agents' into a form that is simultaneously concise and information dense for all decisions in `Decisions`. Thirdly, we implicitly have an "action heat map" over the file which our learning algorithms can leverage to extract information about which lines and character spans in $\mathcal{F}$ are actively being worked on or not.

Attributed Line-by-Line State Representation → State Embeddings From the attributed line-by-line state representation at each timestep t, we generate embeddings for our policy models to ingest using Algorithm 1. The algorithm itself is straightforward, and essentially equates to transforming each component of the attributed line-by-line state representations into numerical features before concatenating them. As a prerequisite for Algorithm 1 we define four "featurizers" that, for each line $\ell \in \mathcal{F}$, maps components of a line's representation to numerical features policy models can consume:

- The line featurizer ψ^{Line} maps plaintext in ℓ to line-level embeddings $l_\ell \in \mathbb{R}^{\gamma_\ell}$ having dimension $\gamma_\ell > 0$. In §4.1.3 we define precisely which line featurizers are implemented in CodeAssist.
- The cursor featurizer ψ^{Cursor} which maps cursor details from ℓ to line-level embeddings $c_\ell \in \mathbb{R}^{\gamma_c}$ having dimension $\gamma_c > 0$. At present the cursor featurizer equates to an identity map on the numerical features already captured in the attributed line-by-line state representation.
- The attribution featurizers ψ^α for $\alpha \in \{H, A\}$ map an agent's operation attribution details from ℓ to line-level embeddings $\tau_\ell^\alpha \in \mathbb{R}^{\gamma_{\tau^\alpha}}$, where dimension $\gamma_{\tau^\alpha} > 0$. At present the attribution featurizers for both agents equates to an identity map on the numerical features already captured in the attributed line-by-line state representation.

Algorithm 1 Generating state embeddings from line-by-line attributed state representations.

$E_t^\alpha \leftarrow \{\}$
for $\ell \in \mathcal{F}_t$ **do**
  $l_\ell \leftarrow \psi^{Line}(\ell)$
  $c_\ell \leftarrow \psi^{Cursor}(\ell)$
  $\tau_\ell^H \leftarrow \psi^H(\ell)$
  $\tau_\ell^A \leftarrow \psi^A(\ell)$
  $e_\ell \leftarrow \texttt{concat}(l_\ell, c_\ell, \tau_\ell^H, \tau_\ell^A)$
  $E_t^\alpha \leftarrow \texttt{stack}(E_t^\alpha, e_\ell)$
end for

After applying Algorithm 1 to the attributed line-by-line state representation at timestep t, we get a line-by-line state embedding $E_t^\alpha \in \mathbb{R}^{L \times \gamma}$ where $\gamma = \gamma_\ell + \gamma_c + \gamma_{\tau^H} + \gamma_{\tau^A}$ and the superscript $\alpha \in \{H, A\}$ indicates whether the embedding was generated for making an assistant vs human decision. In particular, since we assume human then assistant action ordering per timestep, when $\alpha = A$ the line-by-line state

representation used for generating the embedding corresponds to $s_t \cdot a_t^H$ whereas when $\alpha = H$ it corresponds to $s_{t+1} = s_t \cdot a_t^H \cdot a_t^A$. Finally, for completeness, state-action trajectories of length $T > 0$ (an *episode*) define agent-specific episode trajectory embeddings $E^\alpha = \{E_t^\alpha\}_{t \in [T]} \in \mathbb{R}^{T \times L \times \gamma}$ where $\alpha \in \{H, A\}$ denotes whether this embedding tensor is from the perspective of an assistant policy or human policy. In practice episodes play the key role of defining individual "end-to-end" interactions between humans and assistants, where each episode is simply the sequence of human-assistant turns from $t = 0$ until either the task is complete or the human terminates the interaction.

State Embeddings → Decisions & Actions We are now ready to discuss how decisions are chosen by human and assistant policies–π^H and π^A, respectively–which in turn translate to delegate tool-calls that yield actions.

Let $t \in [T]$ be some timestep in a CodeAssist TMG and s_t, a_t^H, (d_t, x_t) be the corresonding state, human action, and assistant decision respectively; assume we have sampled reward parameters θ. As defined in §2 and §3, we know $a_t^H \sim \pi^H(a^H \mid h_t, \theta)$ and $a_t^A = \Phi^A(s_t, d_t, x_t) \sim \pi^A(a_t^A \mid h_t)$. If the human policy is a human model we can mirror the design of the assistant policy and identically define a human actuation map Φ^H, which we will need to do in the future for zero-style self-play training. However, since we are not yet using the human models and it translates $1 - 1$ to assistant models, we will focus our discussion here on the assistant policy model.

Throughout this section we have described how to transform state-action histories[4] into attributed line-by-line state representations and then into the state embedding E_t^A. The assistant policy model (specific architectures introduced in §4.1.3) ingests E_t^A and outputs a decision $\texttt{d} \in \texttt{Decisions}$. In CodeAssist we define a unique bijective mapping from each decision in `Decisions` to a specific tool-prompt pair, where the prompt only differs in t because it includes information about the state s_t. Currently the assistant uses the same coding-tuned LLM and only changes the prompt depending on the decision, but we will loosen this constraint in the near future so that each decision maps to a tool most appropriate for it. From the resultant tool-prompt pair, the assistant calls the tool with the corresponding prompt which yields a unified diff. The actuation map Φ^A in the current implementation can therefore be seen as the act of calling the tool and then applying the resultant unified diff to the file, which is treated as the assistant's action in the shared environment.

Before concluding the discussion, it is interesting to note that when the human policy is an actual human we can define an actuation map $\Phi^H : (\mathcal{S} \times \mathcal{A}^H \times \mathcal{A}^A)^* \times \mathcal{S} \to \mathcal{A}^H$ which models a hierarchical decision-making process in humans. However, unless it becomes relevant for future directions in our endeavors to improve human modeling for zero-style training, we will not explore the idea further in this report as it is out of scope.

4.1.2. Rewards

CodeAssist includes four unique "types" of reward functions, and improving as well as extending the set of reward functions will likely be an ongoing endeavor for the duration of this project. We split the reward functions into two categories depending on the type of signal they provide the policy models–dense reward signals and sparse reward signals. Below we highlight all signals in CodeAssist today and give their default values as an example in their description.

Dense reward signals can be computed at every timestep, and currently include the following.

- **Compilation Delta Rewards:** A positive signal (e.g. +0.75) if the code goes from not compiling to compiling after the assistant turn, a negative signal (e.g. −0.5) if the opposite holds, and a weak

[4] Technically we do not use the literal state-action histories, and only rely on a condensed representation of h_t. This choice is only for simplicity in practice.

positive signal (e.g. +0.1) if the code compiles before and after the assistant turn.
- **Test Delta Rewards:** A positive signal (e.g. +1.0) if the fewer tests passes before the assistant turn as compared to after, as well as a negative signal (e.g. −1.0) if the opposite holds.
- **Interaction Rewards:** A weak positive signal (e.g. +0.25) if the assistant "places" lines in the file during its turn and those lines are left largely unchanged (based on a string similarity threshold, e.g. similarity> 0.7) after the human takes their turn. Similarly, a strong negative signal (e.g. −0.75) when the opposite holds.
- `NO-OP` **Shaping Rewards:** A weak positive signal (e.g. +0.05) when the assistant chooses to `NO-OP` that diminishes as the sequence of uninterrupted `NO-OP` decisions grows. When this sequence of `NO-OP`s hits a predefined count (e.g. 2), the reward becomes a weak negative signal (e.g. −0.05) that grows multiplicatively (e.g. by a factor of 1.5) until a cap (e.g. −5.0).

Sparse reward signals can (generally) only be computed at the end of the episode, and currently include the following.

- **Survival Rewards:** A strong positive signal (e.g. +1.0) if the assistant "places" lines in the file during its turn and those lines are left largely unchanged (based on a string similarity threshold, e.g. similarity> 0.7) until the end of the episode. Similarly, a weak negative signal (e.g. −0.25) when the opposite holds.

There are myriad reward signals we might consider including in the future, and will certainly incorporate more sparse signals in the future. For example, signals based on information about errors from human attempts to submit solutions and how the assistant behaved before/after.

4.1.3. Policy Model Architectures

CodeAssist relies on 3 models serving distinct purposes:

- State embedding model → **Training Optional.** This model is essentially the set of featurizers and their orchestration described in Algorithm 1.
- Assistant policy model → **Trained.** Takes in embeddings E_t^A at each time step t and outputs a decision in `Decisions`. This model has multiple heads we describe below, and they will all become more important in future expansions.
- Human policy model → **Trained.** Very similar to the assistant policy model, but aims to emulate how the human engaged with the coding problem rather than how to assist the human. This model will use a very similar architecture to the assistant policy model, but we will leave a deeper exploration of this model for future expansions when we start experimenting.

We discuss each of these models below and their architectures currently implemented in CodeAssist.

Featurizers CodeAssist released with four line featurizer options included:

- Text Byte MLP → **Trainable.** A simple, shallow MLP that projects direct line-level byte encodings to the appropriate output dimension. **Currently the default featurizer.**
- Character Pooling MLP → **Trainable.** A simple MLP that first pools the line's plaintext over ASCII characters to give a count per character, then passes these counts through two MLP layers that project to the appropriate output dimension.
- Character CNN → **Trainable.** A CNN-based architecture that first encodes the line's characters using a simple mapping, then applies a one-dimensional convolution to that encoding, and finally projects to the appropriate output dimension.

- Frozen LLM Embedder → **Not Trainable.** Calls a frozen, lightweight, and coding-tuned LLM to ingest the line's plaintext. We extract an embedding from its internal layers and project it to the appropriate output dimension.

Special cursor and attribution featurizers are currently not implemented, and these features are simply appended to the line featurizer's output. Once the full line-level embedding is available we pass them through a final fully connected projection layer, which is also trainable.

Assistant Policy Models At each time step t, we assume the assistant policy model receives a state embedding E_t^A. We split the assistant policy into its "backbone" and "heads".

Intuitively model backbones are responsible for outputting a vector per line that "bakes in" information relevant to said line from across the file, which the model heads can then utilize for generating policy outputs. These backbones receive state embeddings E_t^A, mix inter- and intra-line information, and output a tensor of shape $L \times K$ where $K > 0$ is dependent on the backbone architecture we use. In addition, depending on the backbone architecture, an additional output tensor of shape $G \times K$ may be produced that explicitly serves to capture file-level information.

Three backbones are implemented in CodeAssist on release:

- A local-global transformer backbone which utilizes the rich multi-step file-wide information captured by our attributed line-by-line state representation and takes advantage of the fact that the context window should be relatively small since it is directly a function of L. **Currently the default backbone.**
- An LSTM-based backbone that closely matches the design in AssistanceZero (Laidlaw et al., 2025). May struggle with long-range non-local dependencies in the file, but ultimately be the best option when considering very long episodes or multi-file setting.
- A BiGRU-based backbone which essentially mirrors the LSTM-based backbone, but could be useful if we begin struggling with memory usage or latency while using the other backbones.

The model heads ingest the tensors output by the backbone and each head is responsible for producing output that serves a specific purpose. The policy models implemented in CodeAssist include four heads:

- Decision selection head → Picks decision `d` $\in$ `Decisions` that the assistant policy should use. Also, if relevant to `d` (e.g. `Edit Existing Lines`), the decision selection head chooses which lines the resulting tool-prompt pairs should be applied to.
  - Outputs a distribution over `Decisions` and, if relevant, a distribution over $\ell \in \mathcal{F}$.
  - Uses a shallow MLP architecture yielding output at the line-level per decision, and applies some masking as well as normalization to both shape and calibrate the outputs.
- Value head → Estimates how promising the current state is for getting future rewards.
  - Outputs a scalar measuring the "value" of the current state, i.e. the expected reward from taking an action at the current state.
  - Uses a shallow MLP architecture that pools information from across the file.
- Goal head → Infers what the human "cares about" at the current timestep. Currently not wired in.
- Human decision prediction head → Predicts the next human decision (inferred from attributions in the line-by-line state representation) in `Decisions`.
  - Both the outputs and architecture are identical to the decision selection head.

The value and human decision prediction heads will ultimately play key roles for monte carlo tree (MCTS), but currently only serve as auxiliary loss terms for conditioning the policy. In upcoming work we will focus on integrating MCTS variants and zero-style self-play in the training pipeline.

4.1.4. Training Pipeline

As we've discussed throughout this report, the training pipeline in CodeAssist yields two complimentary models–π^H and π^A. Ultimately we want the process to proceed in three stages: (i) inference & episode recording, (ii) supervised pretraining, and (iii) zero-style self-play.

The zero-style training stage is not currently being used in CodeAssist as we work on measuring performance of a supervised PPO baseline and incorporating user feedback based on the initial release on 11/05/2025. Once the CodeAssist app has stabilized and we have a better idea of how best to incorporate zero-style training, we will reintroduce that stage to the pipeline and measure its impact on assistants' performance; at that time we will discuss it in detail in this report.

Episode generation (i.e. inference). Episodes are generated by real humans solving LeetCode problems alongside their assistant policy model.

During inference, as actions are taken by either humans and/or assistants, the state embeddings are generated from attributed line-by-line state representations and ingested by policy models in order to decide which tool-prompt pair the assistant should call to (hopefully) aide in accomplishing the joint goal defined by the problem.

Afterwards, the resultant state-action trajectories from these episodes are stored locally, processed further if needed, and later ingested by policy models for training.

Supervised pretraining. From episodes recorded alongside humans, we extract "supervised" learning signals that are used to pretrain both human and assistant policy models. Although we pretrain both agents' policies on recorded episodes, the method differs for humans vs assistants due to fundamental differences in their learning objectives:

- For human policy pretraining the goal is to learn to imitate human coding behaviours, and the outcome we hope for is a model that realistically makes decisions like the human and picks models that are able to emulate human actions. Currently, since we are not using zero-style training or tree search heuristics like PUCT (Schrittwieser et al., 2020), the human policy being trained will not be used by CodeAssist.
- For assistant policy pretraining the goal is to anchor the policy to reasonable priors based on episodes from actual humans, which will only become more crucial when we start incorporating zero-style training.

Assistant policy pretraining currently involves doing N epochs (40 by default) of supervised PPO on human-recorded episodes. The decision selection head uses a standard PPO clipped surrogate loss whereas the value and human decision prediction heads use mean-squared error against their natural targets, i.e. realized rewards for the value head and inferred human decision for the human decision prediction head.

4.2. Phase 2: Beyond LeetCode-style Tasks

When we move to this phase of the project the training pipeline will be relatively stable and well understood. The aim here will be to move beyond LeetCode-style problems and towards truly open-ended coding tasks in a single file. In this phase we will be able to leverage and explore the full flexibility TMGs provide us, e.g. giving assistants the ability to open a "chat" for users to engage with powerful point-and-shoot agents.

4.3. Phase 3: Arbitrary Multi-File Code Assistance

Upon reaching this phase of the project, the UI/UX and training pipelines will be well-understood and ready to generalize into essentially arbitrary coding tasks. We will begin moving beyond single-file assistance and have to rework various core facets of CodeAssist so that assistants are able to learn in complex multi-file settings.

5. Experiments

Measuring "assistance" is difficult in practice, and much of the progress and iteration in the CodeAssist project will be driven by metrics measured from real users working with their assistants. While building and debugging the initial release of CodeAssist we confirmed that loss curves converged as expected and that assistants seemed to be learning to adapt to the Gensyn team's behaviours, but this is not sufficient to conclude that the assistants are "performant" and it certainly is not sufficient for drawing reliable conclusions as we iterate–the assistants trained with CodeAssist should ideally be able to assist essentially any user with their unique preferences and style, not just the Gensyn team's. With the release of CodeAssist on 11/05/2025 we are opening the project to the public so that we may understand what does and does not work as we methodically add further complexity into the training pipeline.

Once we have had a chance to verify and analyze user metrics we will add a more in-depth discussion to this section. Our focus between now and the next update to this report will be four pronged:

1. Incorporate user feedback into features or roadmap items.
2. Measure assistance performance proxies, e.g. number of turns to task success as a function of training statistics, value head outputs vs realized rewards, and assistant line survival rates.
3. Analyze assistant behaviour distributions as a function of training, e.g. distribution shift of `NO-OP` decisions as a function of number of episodes used in training.
4. Understand recorded episode distributions, e.g. episode duration and assistant latency statistics.

6. Conclusion

In this report we lay the groundwork for our commitment to maintaining a "living paper" for CodeAssist; a research report that will get periodically updated with results, insights, etc. after each big update made to the CodeAssist app. We are taking a unique approach to this research–an approach anchored in transparency. We will continue to build and iterate on the project completely in the open as we believe these principles must be entrenched in AI development so that we may all flourish, and not just a select few big companies. By breaking down silos and building together we can accelerate research progress without compromising on transparency, safety, and real-world utility of AI systems.

There are countless ways to contribute to CodeAssist, but here are three concrete routes:

- If you are non-technical, you can help us understand what users want and what actually improves assistants simply by using the app and providing feedback–if you do not know how to code, join us on Discord[5] and our wonderful community will help you get started!
- If you have experience working on software or contributing to open-source projects, you can help improve the app by contributing on GitHub[6]! The project is sprawling and has numerous features, improvements, etc. that will always need more eyes and hands on them.

[5] https://discord.gg/gensyn
[6] https://github.com/gensyn-ai/codeassist

- If you are an experienced researcher with relevant expertise, you are welcome to reach out and we can talk about possible collaborations! There are more open questions than hours in the day, so these sorts of collaborations will be essential as we make progress.

For more information on how to run the CodeAssist app you can find thorough documentation in the README on GitHub or directly in Gensyn's documentation hub[7].

References

Yuntao Bai, Andy Jones, Kamal Ndousse, Amanda Askell, Anna Chen, Nova DasSarma, Dawn Drain, Stanislav Fort, Deep Ganguli, Tom Henighan, et al. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*, 2022.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Alan Fern, Sriraam Natarajan, Kshitij Judah, and Prasad Tadepalli. A decision-theoretic model of assistance. *J. Artif. Int. Res.*, 50(1):71–104, May 2014. ISSN 1076-9757.

Dylan Hadfield-Menell, Anca Dragan, Pieter Abbeel, and Stuart Russell. Cooperative inverse reinforcement learning, 2016. URL https://arxiv.org/abs/1606.03137.

Cassidy Laidlaw, Eli Bronstein, Timothy Guo, Dylan Feng, Lukas Berglund, Justin Svegliato, Stuart Russell, and Anca Dragan. Assistancezero: Scalably solving assistance games, 2025. URL https://arxiv.org/abs/2504.07091.

Leon Lang, Davis Foote, Stuart J Russell, Anca Dragan, Erik Jenner, and Scott Emmons. When your ais deceive you: Challenges of partial observability in reinforcement learning from human feedback. *Advances in Neural Information Processing Systems*, 37:93240–93299, 2024.

Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. In William W. Cohen and Haym Hirsh, editors, *Machine Learning Proceedings 1994*, pages 157–163. Morgan Kaufmann, San Francisco (CA), 1994. ISBN 978-1-55860-335-6. doi: https://doi.org/10.1016/B978-1-55860-335-6.50027-1. URL https://www.sciencedirect.com/science/article/pii/B9781558603356500271.

Omid Madani, Steve Hanks, and Anne Condon. On the undecidability of probabilistic planning and related stochastic optimization problems. *Artificial Intelligence*, 147(1):5–34, 2003. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(02)00378-8. URL https://www.sciencedirect.com/science/article/pii/S0004370202003788. Planning with Uncertainty and Incomplete Information.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

Christos H. Papadimitriou and John N. Tsitsiklis. The complexity of markov decision processes. *Math. Oper. Res.*, 12(3):441–450, August 1987. ISSN 0364-765X.

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

Lior Shani, Aviv Rosenberg, Asaf Cassel, Oran Lang, Daniele Calandriello, Avital Zipori, Hila Noga, Orgad Keller, Bilal Piot, Idan Szpektor, et al. Multi-turn reinforcement learning with preference human feedback. *Advances in Neural Information Processing Systems*, 37:118953–118993, 2024.

[7] https://docs.gensyn.ai/testnet/codeassist

L. S. Shapley. Stochastic games*. *Proceedings of the National Academy of Sciences*, 39(10):1095–1100, 1953. doi: 10.1073/pnas.39.10.1095. URL https://www.pnas.org/doi/abs/10.1073/pnas.39.10.1095.

Mrinank Sharma, Meg Tong, Tomasz Korbak, David Duvenaud, Amanda Askell, Samuel R Bowman, Newton Cheng, Esin Durmus, Zac Hatfield-Dodds, Scott R Johnston, et al. Towards understanding sycophancy in language models. *arXiv preprint arXiv:2310.13548*, 2023.

Marcus Williams, Micah Carroll, Adhyyan Narang, Constantin Weisser, Brendan Murphy, and Anca Dragan. On targeted manipulation and deception when optimizing llms for user feedback. *arXiv preprint arXiv:2411.02306*, 2024.

Yuzi Yan, Xingzhou Lou, Jialian Li, Yiping Zhang, Jian Xie, Chao Yu, Yu Wang, Dong Yan, and Yuan Shen. Reward-robust rlhf in llms. *arXiv preprint arXiv:2409.15360*, 2024.

Michael JQ Zhang, Zhilin Wang, Jena D Hwang, Yi Dong, Olivier Delalleau, Yejin Choi, Eunsol Choi, Xiang Ren, and Valentina Pyatkin. Diverging preferences: When do annotators disagree and do models know? *arXiv preprint arXiv:2410.14632*, 2024.