# Training LLM Agents to Empower Humans

**Evan Ellis** [1] **Vivek Myers** [1] **Jens Tuyls** [2] **Sergey Levine** [1] **Anca Dragan** [1] **Benjamin Eysenbach** [2]

## Abstract

Assistive agents should not only take actions on behalf of a human, but also step out of the way and cede control when there are important decisions to be made. However, current methods for building assistive agents, whether via mimicking expert humans or via RL finetuning on an inferred reward, often encourage agents to complete tasks on their own rather than truly *assisting* the human achieve her objectives. We propose a new approach to tuning assistive language models based on maximizing the human's *empowerment*, their ability to effect desired changes in the environment. Our empowerment-maximizing method, `Empower`, only requires offline examples of human-written text, providing a self-supervised method for fine-tuning language models to better assist humans. To study the efficacy of our approach, we conducted an 18-person user study comparing our empowerment assistant with a strong baseline. Participants preferred our assistant 78% of the time ($p = 0.015$), with a 31% higher acceptance rate. With this empowerment objective, we provide a framework for useful aligned AI agents without the need for human feedback or verifiable rewards.

## 1 Introduction

Software developers today face a challenge when using LLM coding agents: code suggestions start out helpful but then start implementing the wrong functions. Often an assistant will suggest a large block of code, the user accepts it, and then she has to spend time fixing the parts it got wrong, such as an incorrect assumption. How can we develop coding assistants that still produce helpful generations, but also know to stop their generations at critical junctures? While this problem is especially salient for coding assistants (the focus of this paper), such problems are likely to recur in applications from assistive robotics to interacting with autonomous web agents (Chen et al., 2021; Trivedi et al., 2024).

Optimizing for helpfulness is challenging. Gathering explicit human labels is expensive and time-consuming. Additionally, it is unclear how this sort of helpfulness can be learned from traces of a human expert — the problem is not that generations are unrealistic, but rather that they may be solving a problem that is different from what the user intends. One approach to this problem is for agents to ask clarifying questions to better infer the intentions of human users. However, this style of assistance requires interrupting the user, possibly impeding their flow and making the interaction feel burdensome. In many situations, it is desirable to have an assistant which does not rely on querying the user.

The key insight in this paper is to train assistive agents to *empower* human users, a training method that does not require that the agents know the human's underlying intention. Intuitively, empowerment refers to an agent's ability to effect changes in the environment. Rather than asking a human for explicit feedback, the LLM will automatically assess the usefulness of its actions by estimating whether they enable a human to solve more tasks more quickly. In coding contexts, empowerment might correspond to implementing helper functions, writing boilerplate, or wrapping up lines of code. Mathematically, empowerment is defined by a mutual information that measures the degree of control that an agent's actions exert on states that occur in the future (Klyubin et al., 2005). The assistive setting, where there are two agents (an assistant and a human), requires a more nuanced definition of empowerment: we aim to empower the *human* agent, enabling the human user's actions to exert a larger influence over future outcomes (Du et al., 2020; Myers et al., 2024). An agent that maximizes the human's empowerment will help them reach goals more effectively, without assuming any prior reward structure. Similar empowerment objectives are used in psychology to explain certain facets of human learning (Gopnik, 2024).

We use code generation as a context for studying empowerment-maximizing assistants because it is one of the few real-world applications today where humans regularly interact with assistive agents (e.g., Github Copilot, Claude

[1]University of California, Berkeley [2]Princeton University. Correspondence to: Evan Ellis <evan.ellis@berkeley.edu>.

*Proceedings of the $43^{rd}$ International Conference on Machine Learning*, Seoul, South Korea. PMLR 306, 2026. Copyright 2026 by the author(s).

Code, Cursor). It is also an appealing starting point because preexisting datasets (Jain et al., 2024) allow us to measure the efficacy of empowerment maximization in a rigorous way.

**Contributions.** In this paper, we derive a practical and scalable algorithm for training LLM agents to maximize empowerment, and demonstrate its effectiveness in the code generation setting. Specifically, our work makes the following contributions:

1. **`Empower` method (Section 4.2).** We propose `Empower`, a method for aligning LLM agents to work with humans based on the objective of maximizing effective empowerment. Our method provides a proof-of-concept for training an LLM agent to maximize a human's empowerment.
2. **Simulated results (Section 5.2).** Experimental results with a Gemma-3-27B-it (Team et al., 2025) human on LiveCodeBench (Jain et al., 2024) show that our method leads to a higher Pass@1 without explicit human feedback. A Llama-3.1-8B-Instruct model trained with `Empower` doubles the Pass@1 rate compared to the strongest baseline.
3. **User study (Section 5.3).** We demonstrate in a user study that human coders prefer our empowerment assistant over a baseline. Participants preferred our assistant 78% of the time ($p = 0.015$), and accepted our assistant's suggestions 31% more often than the baseline's suggestions ($p = 0.0002$). Additionally, participants tended to delete 26% fewer characters they had accepted from our method than from the baseline ($p = 0.012$). This amounts to a stronger assistant that is more likely to suggest code the user will accept and actually use.

Taken together, our results demonstrate that LLM assistants can be trained without receiving feedback or interaction from humans by reasoning about how their actions might enable humans to complete more tasks more quickly.

## 2 Related Work

Past work has studied empowerment in the context of intrinsic motivation and reinforcement learning.

**Empowerment.** Informally, empowerment quantifies the influence an agent's actions have over outcomes in the environment. Empowerment has traditionally been defined as the channel capacity between a sequence of actions and the following state (Abel et al., 2025; Klyubin et al., 2005; 2008; Salge et al., 2014). Empowerment objectives have been used in single-agent reinforcement learning to enable intrinsic motivation and exploration (Choi et al., 2021; de Abril and Kanai, 2018; Mohamed and Jimenez Rezende, 2015; Eysenbach et al., 2018). More recently, empowerment has been explored for collaborative settings, where a robot assistant learns to maximize the empowerment of a human user. Du et al. (2020) propose the AvE algorithm which enables assistance by computing empowerment with random rollouts. Myers et al. (2024) show how a contrastive objective can scale empowerment maximization. However, these prior works typically use simple gridworld-like or video game-like environments. On larger web agent benchmarks, Song et al. (2025) found that the effective empowerment is highly correlated with LLM task performance. Their work focused on evaluating the correlation between effective empowerment and reward, rather than on optimizing LLM agents with an empowerment objective. To the best of our knowledge, ours is the first to apply empowerment to train LLM agents, applying it at scale to a realistic coding task. This is enabled by our insight that LLM uncertainty can be used to identify key decision points, and we can empower people by helping them reach those points.

**Learning from Human Preferences.** Many methods attempt to align AI agents by updating the agent with human preference information. Christiano et al. (2017) used an online stream of human preferences to train a reward model concurrently with an actor-critic policy. This method (RLHF) was later adapted to align LLMs to human preferences, enabling conversational agents like InstructGPT (Stiennon et al., 2020; Ouyang et al., 2022; Dong et al., 2023). These methods often use PPO or related policy-gradient methods to fine-tune a pre-trained LLM (Schulman et al., 2017; Sutton and Barto, 2018; DeepSeek-AI et al., 2025). Learning and optimizing a reward model can be expensive or unstable, leading to alternative methods like DPO (Rafailov et al., 2024) and IPL (Hejna and Sadigh, 2023) that directly optimize the policy to match human preferences without an intermediate reward model. Training with human feedback faces key limitations: human values may be difficult to represent with reward functions (Casper et al., 2023), they may change over time (Carroll et al., 2024), and optimizing them may lead to misaligned behaviors like power seeking and manipulation (Williams et al., 2024). Our empowerment objective offers an alternative strategy for aligning LLMs which instead completes tasks for the human that are obvious and general, rather than aligning assistance to a set of preferences.

**Self-Supervision for LLMs.** There is existing precedent for having LLMs provide their own feedback. Self-critiquing methods are common in mathematical and logical reasoning applications (Yao et al., 2024; Madaan et al., 2024; Shinn et al., 2024), as are methods that leverage the model itself to provide a learning signal for self improvement (Huang et al., 2025; 2023; Pang et al., 2023; Yuan et al., 2024). In contrast, our work places the human back at the center: rather than optimizing an LLM to produce text that another LLM thinks is correct, we optimize an LLM to produce code that enables a human

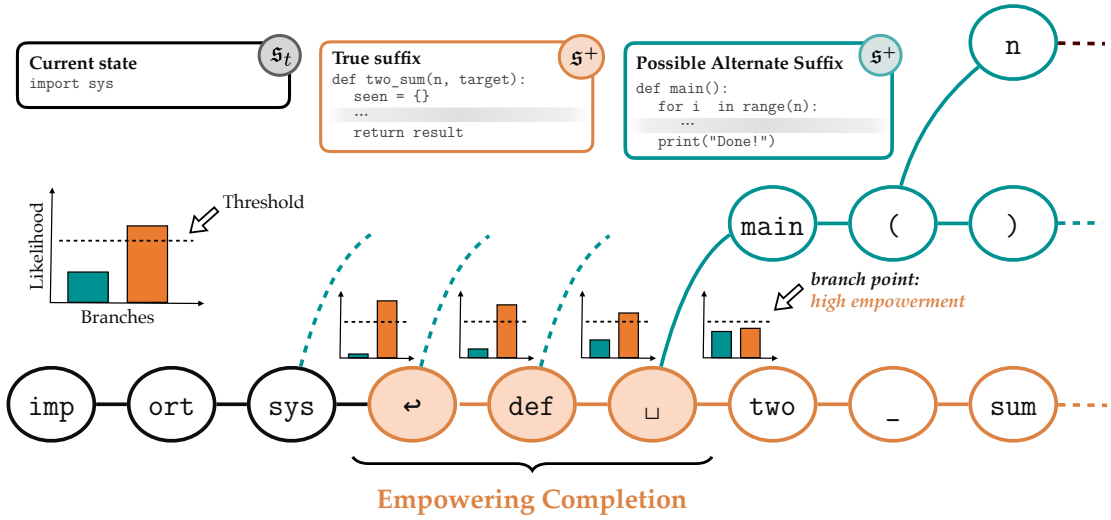


*Figure 1.* **Training assistive agents via `Empower`.** An LLM generates the cumulative likelihood of the suffix, shown below each token. Empowering completions are selected as the longest suffix where the cumulative likelihood is greater than a threshold. This trains the assistant to complete text up to a decision point. Then, the human will have more choices about where to take the program, so their next action is *empowered*.

to solve more tasks more quickly.

**Assistive Agents.** One mathematical framework for assistive agents is the assistance game (Hadfield-Menell et al., 2016). The assistance game extends the standard Markov decision process (MDP) definition to include a human and a robot agent which interact in a shared environment to maximize a joint reward that is only known to the human. The agent must combine inference of the human rewards or goals with reinforcement learning to optimize the inferred objective (Carroll et al., 2019; 2024; Hadfield-Menell et al., 2017; Laidlaw et al., 2024). Methods that learn from human preferences (Christiano et al., 2017; Rafailov et al., 2024) can be seen as special cases of the assistance game where the human's actions only exist to provide information about the reward to the agent. Other works within the assistive setting focus on task completion with humans in the loop, where the robot assistants need to learn when to ask humans for help and what to ask them for (Ren et al., 2023; Ramrakhya et al., 2025; Mullen and Manocha, 2024). Empowerment methods are a special case of assistance games, where the empowerment objective acts as a proxy for the human's reward function (Myers et al., 2024). Our work builds on this foundation to study how empowerment methods might be scaled to align LLMs.

Our contribution is to connect empowerment to the problem of aligning LLM agents to human users, showing that the empowerment objective can provide a scalable self-supervised learning signal for an LLM agent aiding a human in an assistive setting.

## 3 Preliminaries

We cast the problem of an LLM agent assisting a human as a Markov decision process (MDP). The state is the program text at the given timestep. At each state, the LLM agent action suggests a piece of text to append to the conversation. The human agent first chooses to `ACCEPT` or `REJECT` the suggestion, or `FINISH` writing the program. Then, unless they choose to `FINISH`, they will append some number of tokens.

**Notation.** In this assistance MDP, the human policy $\pi_{\mathbf{H}}$ selects an action $a^{\mathbf{H}} \in \mathcal{A}_{\mathbf{H}}$, and the LLM agent's policy $\pi_{\mathbf{R}}$ selects an action $a^{\mathbf{R}} \in \mathcal{A}_{\mathbf{R}}$ to complete the code snippet. Let $\mathcal{L}$ be the set of possible tokens and $\mathcal{L}^*$ be the set of all strings consisting of these tokens. The human's actions are $\mathcal{A}_{\mathbf{H}} = (\{\texttt{ACCEPT}, \texttt{REJECT}\} \times \mathcal{L}) \cup \{\texttt{FINISH}\}$, where `ACCEPT` appends the LLM agent's suggestion to the conversation followed by the human's own text, `REJECT` does not append the suggestion and only appends the human's text, and `FINISH` ends the episode. If the human does not choose `FINISH`, they will write one token. The LLM agent's actions are $\mathcal{A}_{\mathbf{R}} = \mathcal{L}^*$, where the agent suggests any sequence of tokens to append to the conversation. Note that $\ell_{i:j}$ indexes the tokens from $i$ to $j$ inclusive. We will use $s_t = \ell_{1:n}$ to indicate that the current state is $n$ tokens. The assistant's suggestion is $a_t^{\mathbf{R}} = \ell_{n+1:n+i}$, where $i$ is the number of tokens it proposed. If the human accepts the suggestion, then the human will write $\ell_{n+i+1}$. If the human rejects the suggestion, then they will write $\ell_{n+1}$.

We will define random variables $\mathfrak{s}_t$ and $\mathfrak{a}_t^{\mathbf{H}}$ to denote the state and human action at time step $t$. Similarly, in state $s_t = \ell_{1:n}$, if the human does not accept the assistant's

suggestion, and let $\ell^{\mathbf{H}}_{n+1}$ represent the random variable of the next token that the human writes. Let $\ell^{+}$ be a random variable over possible future text. Additionally, $\hat{\pi}(\ell_{n+1:n+i} \mid \ell_{1:n})$ denotes a conditional probability distribution over possible completions. We will choose this to be a pre-trained LLM.

**Empowerment.** We will define empowerment using the mutual information $I(\cdot;\cdot)$ between two random variables. In a single-agent MDP, empowerment is defined by Klyubin et al. (2005) as the channel capacity between a sequence of $n$ actions and the resulting state $n$ steps into the future:

$$C\big(p(\mathfrak{s}_{t+n} \mid \mathfrak{a}^n_t, \mathfrak{s}_t)\big) \triangleq \max_{p(a^n_t \mid \mathfrak{s}_t)} I(\mathfrak{a}^n_t; \mathfrak{s}_{t+n} \mid \mathfrak{s}_t). \quad (1)$$

Informally, this objective states that empowerment is the maximal degree to which the next $n$ actions $\mathfrak{a}^n_t$ selected in the MDP can impact the resulting state $\mathfrak{s}_{t+n}$. This objective is intuitively appealing because it provides a mathematical way of quantifying whether an assistive agent's actions are useful, without knowing the humans reward function. However, it is challenging to use this objective in practice because *(i)* it involves optimizing over a sequence of actions, and *(ii)* mutual information is non-trivial to compute in high-dimensional environments or over long horizons.

# 4 Maximizing Empowerment over Language

In this section we discuss how to train empowering assistants in the language domain. Empowerment is a useful objective for assistance because it helps people quickly reach states where they have many choices, so it takes broadly useful actions that are helpful for the most people. This leads to a more natural type of assistance that does not make assumptions about, or even try to infer, the human's goal.

Section 4.1 introduces the core problem of constructing a training dataset for the assistant with only offline human data. Section 4.2 describes `Empower`, our practical method for choosing completions to train the assistant. We take the longest completion where the cumulative likelihood of the completion, as judged by an LLM, is above a certain threshold. Section 4.3 shows that, under certain assumptions, our method is estimating the empowerment of a completion. Using this estimate, we train the LLM assistant to complete text that has a low empowerment — text that is predictable — so that the human does not have to write it. Instead, the human can focus on important design decisions, rather than boilerplate code.

## 4.1 Using Offline Data

Formally, given an offline dataset of text written by the human, $\ell_{1:N} \sim \pi_{\mathbf{H}}$, the challenge is in choosing the best state-action pairs for finetuning the assistant. One approach would have an assistant model generate a proposed completion and then use some external feedback, such as whether the human would accept the suggestion, to score its quality. However, we do not assume access to human preference data, so the training signal must come from the text the human wrote. Therefore, we will train the assistant to output the same text that the human wrote in the dataset. This removes the need for `ACCEPT`/`REJECT` feedback, because the assistant proposes text that the human actually wrote, which we assume would be accepted. Formally, we train the assistant to output $\ell_{n+1:n+i}$ when it is given $\ell_{1:n}$ as the state. For each piece of text in the dataset, we sample a single state $\ell_{1:n}$ by uniformly sampling $t \in [1, N]$. The difficulty lies in choosing the appropriate length of completion to train on.

## 4.2 Our Algorithm: `Empower`

When the human writes boilerplate code, they have a low empowerment because their actions are easily predicted, so they carry little information about the future. To empower the human, an assistant should be trained to complete this predictable text so that the human does not have to. Our insight is that we can use an LLM, $\hat{\pi}$, to estimate how likely a completion is. We therefore propose the following algorithm to choose completions to train our assistant on:

$$i^* = \arg\max_i \big\{ i : -\log \hat{\pi}(\ell_{n+1:n+i} \mid \ell_{1:n}) < \eta \big\}. \quad (2)$$

This optimization chooses the largest completion length, $i$, such that the negative log likelihood of that completion as judged by an LLM is below a threshold $\eta$ which we choose. This can equivalently be viewed as choosing the longest completion length, $i$, where the cumulative likelihood of the completion is greater than $2^{-\eta}$. We write the optimization with a negative log likelihood to highlight that it is a one-sample estimate of the entropy. This mathematically connects our method to empowerment, which we will explain further in Section 4.3.

During training, we first sample a program from an offline dataset, then sample a prefix of that program which becomes the state $\ell_{1:n}$. Any suffix is a possible completion. We train on the suffix $\ell_{n+1:n+i^*}$ chosen by Eq. (2). Intuitively, we are training the assistant on obvious completions — those that the LLM thinks are likely — thereby leaving the human to write more impactful text in the future. We summarize our method in Algorithm 1, and show an illustration in Fig. 1.

## 4.3 Connection to effective empowerment

Under some assumptions, our algorithm can be viewed as training the assistant to suggest text that would have a low empowerment for the human to write. We will analyze the user's empowerment in the no-assistant setting where they append a single token at each step. We use the *effective empowerment* objective (Myers et al., 2024), which provides a computationally-tractable alternative to the canonical em-

**Algorithm 1:** Logit Threshold Empowerment (`Empower`)

**Input:** A text document $\ell_{1:N}$ with sampled prefix $\ell_{1:n}$

**Input:** An LLM $\pi$ to estimate cumulative likelihood

**Output:** Empowering suggestion $\ell_{n+1:n+i^*}$ for state $\ell_{1:n}$

**for** $i \in \{1 \dots N\}$ **do** ▷ *loop through the possible completion lengths*
  $\hat{H} \leftarrow -\log \pi(\ell_{n+1:n+i} \mid \ell_{1:n})$ ▷ *compute the one-sample entropy*
  **if** $\hat{H} > \eta$ **then** ▷ *check if estimated entropy exceeds threshold*
    **return** $\ell_{n+1:n+i-1}$ ▷ *return the last index that was below threshold*
**return** $\ell_{n+1:N}$ ▷ *entropy is always within bounds, so return rest of program*

powerment objective (Klyubin et al., 2005) (see Eq. (1)). Effective empowerment is defined with respect to a specific policy, $\pi_{\mathbf{H}}$, and a future state $\ell^+$. In this work, we take $\ell^+$ to be the final program. We define the effective empowerment at a state $\ell_{1:n}$ as:

$$\mathcal{E}(\pi_{\mathbf{H}}, \ell_{1:n}) \triangleq I(\ell^{\mathbf{H}}_{n+1}; \ell^+ \mid \ell_{1:n}). \tag{3}$$

This is a special case of (Myers et al., 2024) with $\gamma = 0$. This objective measures the impact that the human's action, $\ell^{\mathbf{H}}_{n+1}$, has on their future state. We can rewrite the mutual information with an entropy:

$$I(\ell^{\mathbf{H}}_{n+1}; \ell^+ \mid \ell_{1:n}) = H(\ell^{\mathbf{H}}_{n+1} \mid \ell_{1:n}) - H(\ell^{\mathbf{H}}_{n+1} \mid \ell^+, \ell_{1:n}) \tag{4}$$

Our core observation is that in a deterministic environment where each state contains the information from every previous state, $H(\ell^{\mathbf{H}}_{n+1} \mid \ell^+, \ell_{1:n}) = 0$. Using this, we focus our attention on estimating $H(\ell^{\mathbf{H}}_{n+1} \mid \ell_{1:n})$. Computing this entropy exactly requires knowing the true human policy, $\pi_{\mathbf{H}}(\ell^{\mathbf{H}}_{n+1} \mid \ell_{1:n})$, which we do not have access to. Instead, we assume access to another likelihood estimator, $\hat{\pi}(\ell^{\mathbf{H}}_{n+1} \mid \ell_{1:n})$, which can approximate the human's marginal likelihood of any action at a given state. Then we can approximate the human's marginal entropy $H(\ell^{\mathbf{H}}_{n+1} \mid \ell_{1:n})$ by sampling an action from the human and using a one-sample monte carlo estimate:

$$\hat{H}(\ell^{\mathbf{H}}_{n+1} \mid \ell_{1:n}) \approx -\log \hat{\pi}(\ell^{\mathbf{H}}_{n+1} \mid \ell_{1:n}). \tag{5}$$

In practice, we choose our human entropy estimator $\hat{\pi}$ to be a pre-trained LLM. Our empowerment estimate becomes:

$$\mathcal{E}(\pi_{\mathbf{H}}, \ell_{1:n}) \approx -\log \hat{\pi}(\ell^{\mathbf{H}}_{n+1} \mid \ell_{1:n}). \tag{6}$$

While this is a rough approximation of the entropy, it works well in practice for the purpose of choosing empowering completions, and is simple to implement. This approximation does not capture the human's ability to control the environment, which would be measured in the $H(\ell^{\mathbf{H}}_{n+1} \mid \ell^+, \ell_{1:n})$ term of Eq. (4). In environments without full history, our approximation still provides an upper bound on the user's empowerment. Under these assumptions, the algorithm described by Eq. (2) can be seen as training an assistant to complete text which is *predictable*, and therefore would not be empowering for the human to write.

Another approach to maximizing effective empowerment is to use contrastive classifiers to approximate the conditional mutual information (Myers et al., 2024). While this works for traditional RL gym environments, it does not scale well because the contrastive classifiers are heavily dependent on training with millions of samples of online user interaction, and do not transfer if the user behavior changes. Additionally, this contrastive objective is difficult to analyze, because it can converge on a definition of mutual information that is not intuitively what we mean when we say "empowerment." Finally, training an LLM to maximize this empowerment objective leads it to diverge significantly from the base model, which can cause off-policy behavior that the contrastive empowerment model believes is novel, and so rewards highly. In contrast, our approach is interpretable and tractable for LLM-scale models because of our observation that a simple logit threshold calculation is actually a better objective for creating an empowering assistant in text environments.

We highlight two design decisions in our approximation of effective empowerment.

**Misalignment Entropy.** The first is our decision to not separate misalignment entropy from logical entropy. It is difficult to distinguish between tokens with high entropy because many options lead to the same programmatic behavior and tokens with high entropy because they make an important decision about the following program. Our method treats these scenarios as equals if the log probabilities are equivalent. One future direction is to ask a model whether a token represents an important decision or not, and train until that point.

**Longest Completion.** The second is the decision to train on the longest completion where the estimated empowerment is below a threshold. An alternative approach is to split each code file in the training corpus into the set of completions that would maximize the average empowerment of the human. However, while this would increase the average empowerment on the training set, it is very reliant on the hu-

man taking the exact actions they did in the training set. We believe that our approach is effective because it is simple, interpretable, and avoids maximization bias from multi-step backups. Future work could explore non-monotonic selection like skipping high-entropy tokens to reach low-entropy ones beyond them or more advanced sequence selection policies.

### 4.4 Potential LLM Miscalibration

We use a pretrained LLM as an estimator of the human entropy, differing from (Myers et al., 2024) which relies on online interaction with the human to learn an empowerment model. While our method reduces the variance of the estimator by removing the moving target training of an empowerment model, we potentially add bias if the LLM is not calibrated to the user. Prior work has found that in some settings LLMs are good estimators of human variability, while in other tasks they may overestimate it (Giulianelli et al., 2023). To make our results robust against miscalibration, we include results using three different LLMs as likelihood estimators in Section 5.2, and also ablate the choice of the threshold, $\eta$. Future work might investigate if additional calibration techniques, such as those suggested by (Xie et al., 2024), improve the quality of the trained assistant.

## 5 Experiments: Code Generation

Our experiments apply the empowerment framework discussed in Section 4 to the task of code generation. In Section 5.1, we describe our experiment setup. We then evaluate our method in a novel simulated setup using LiveCodeBench (Section 5.2), after which we validate our findings in the real world by running an 18-person double-blinded human study (Section 5.3).

### 5.1 Experiment setup

**Datasets.** We train all models using a dataset of 4,138 unique questions from Codeforces[1], each of which is paired with one attempted solution by Gemma-3-27B-it (Team et al., 2025). We do not filter the dataset for success on the testcases.

**Models.** We use Llama-3.1-8B-Instruct (Grattafiori et al., 2024), Qwen3-8B (Yang et al., 2025), and Qwen3-14B (Yang et al., 2025) as assistant models. For the simulated setting, we use Gemma-3-27B-it (Team et al., 2025) as the human model. The prompts we use are provided in Section D. We use all models with their default sampling parameters.

[1] https://huggingface.co/datasets/MatrixStudio/Codeforces-Python-Submissions

**Baselines.** We compare against both trained and untrained baselines. **(1) SFT-N** finetunes the assistant on the next $N$ tokens that the human wrote in a particular state, followed by a stop token. We evaluate SFT-10 and SFT-20. **(2) SFT-RAND** trains on random human completions between 1 and 30 tokens long to avoid biasing too much towards a specific completion length. **(3) Base** is simply the base assistant model without any training or restrictions on top. **(4) Base-N** is the same as Base, but we cap the suggestion length at N tokens. We include this baseline since we hypothesize that shorter completion lengths are more likely to be accepted. We evaluate Base-10 and Base-20. **(5) RFT** is trained with Rejection Fine-Tuning (RFT) the base model on a single round of accepted suggestions to Codeforces problems (Yuan et al., 2023).

**Our Method.** Our method, `Empower`, trains on completions returned from Algorithm 1, which we run on all completions in the training dataset before the start of training. We use the untrained base assistant model as our likelihood estimator, $\hat{\pi}$. Crucially, we do not provide the likelihood model access to the relevant Codeforces problem, only the text in the state (i.e. the completion tokens written so far).

### 5.2 Evaluating empowerment in a simulated setup

To evaluate the empowerment assistant with a simulated human, we adopt the MDP structure described in Section 3 where the assistant proposes suggested code completions which the human may accept or reject, and then append their own code. We limit the human action size to 10 tokens and the number of rounds of human and assistant actions per problem to 50. We evaluate on LiveCodeBench (Jain et al., 2024) release #6, a benchmark of competitive programming problems that is regularly updated to avoid contamination.

**Evaluation Metrics.** To evaluate the performance of `Empower` compared to the baselines, we propose the following three different evaluation metrics. **(1) Pass@1** measures the success rate of the generated code snippets by evaluating them on the problem testcases, counting a success only if all of the testcases pass. **(2) Acceptance rate** provides a measure of the human's preference for one assistant's suggestions over another's.

Our third metric is **(3) Discounted Pass Rate (DPR).** A higher acceptance rate is not always beneficial if the suggested completions are not more helpful. Occasionally an assistant will propose a completion which looks good, but actually introduces a bug or confuses the human, leading to a lower pass@1. Similarly, a lower acceptance rate can lead to a higher pass@1, making an assistant appear better even though the real gain in performance is from the human solving the problem on their own. Therefore, we introduce a new metric which we call the *Discounted Pass Rate* (DPR), which is a better measure of good assistance

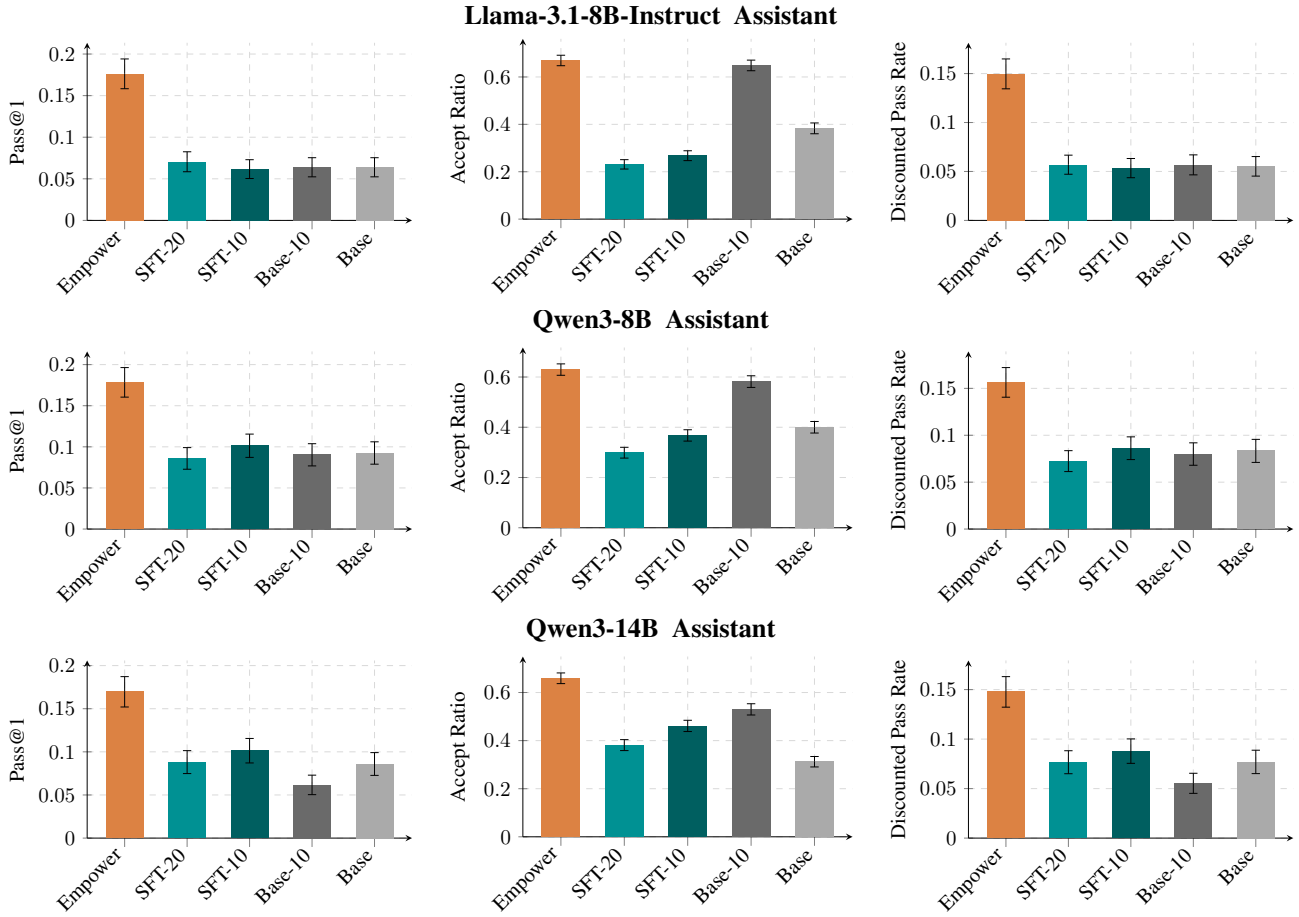


*Figure 2.* **Assistant results with Gemma-3-27B-it as the human model.** We evaluate on 554 LiveCodeBench problems. We find `Empower` to outperform all baselines in terms of pass@1 and DPR. Error bars show standard errors.

*Table 1.* Baselines and $\eta$ ablations with Gemma-3-27B-it as the human model.

| **Base Model** | **Name** | **Pass@1** (↑) | **Accept Ratio** (↑) | **Discounted Pass Rate** (↑) |
|---|---|---|---|---|
| GPT-5-mini | GPT-5-mini | **0.249** $^{(\pm 0.020)}$ | **0.699** $^{(\pm 0.021)}$ | **0.203** $^{(\pm 0.017)}$ |
| Llama3.1-8B Instruct | RFT | 0.057 $^{(\pm 0.011)}$ | 0.384 $^{(\pm 0.023)}$ | 0.049 $^{(\pm 0.009)}$ |
| Llama3.1-8B Instruct | Base-10 | 0.064 $^{(\pm 0.011)}$ | 0.649 $^{(\pm 0.022)}$ | 0.057 $^{(\pm 0.010)}$ |
| Llama3.1-8B Instruct | `Empower` $\eta = 0.32$ | **0.176** $^{(\pm 0.018)}$ | **0.670** $^{(\pm 0.022)}$ | **0.150** $^{(\pm 0.015)}$ |
| Llama3.1-8B Instruct | `Empower` $\eta = 0.5$ | 0.174 $^{(\pm 0.018)}$ | 0.627 $^{(\pm 0.023)}$ | 0.149 $^{(\pm 0.015)}$ |
| Llama3.1-8B Instruct | `Empower` $\eta = 1$ | 0.132 $^{(\pm 0.016)}$ | 0.537 $^{(\pm 0.023)}$ | 0.116 $^{(\pm 0.014)}$ |
| Llama3.1-8B Instruct | `Empower` $\eta = 2$ | 0.117 $^{(\pm 0.015)}$ | 0.524 $^{(\pm 0.023)}$ | 0.104 $^{(\pm 0.014)}$ |
| Llama3.1-8B Instruct | `Empower` $\eta = 4$ | 0.084 $^{(\pm 0.013)}$ | 0.473 $^{(\pm 0.023)}$ | 0.073 $^{(\pm 0.011)}$ |

because it accounts for both the pass rate and the amount of text the human had to read and write to get to a successful program. The DPR for a particular solution is defined as:

$$\text{DPR} = 1_{\text{Correct Solution}} \cdot \gamma^{\alpha \cdot \text{Tokens Read} + \beta \cdot \text{Tokens Written}} \quad (7)$$

The constant $\alpha$ specifies how "difficult" it is for the human to verify text that the assistant has suggested. Similarly, $\beta$ specifies how "difficult" it is for the human to write text on their own. Under this metric, the best assistant will help the human have the highest pass rate, while only suggesting completions which are most likely to be accepted and bring the program closest to its conclusion. We use $\gamma = 0.999$, $\alpha = 0.1$, and $\beta = 0.5$ to represent that it is often more difficult to generate than to verify, as well as to prevent the DPR of a long but correct solution from approaching 0.

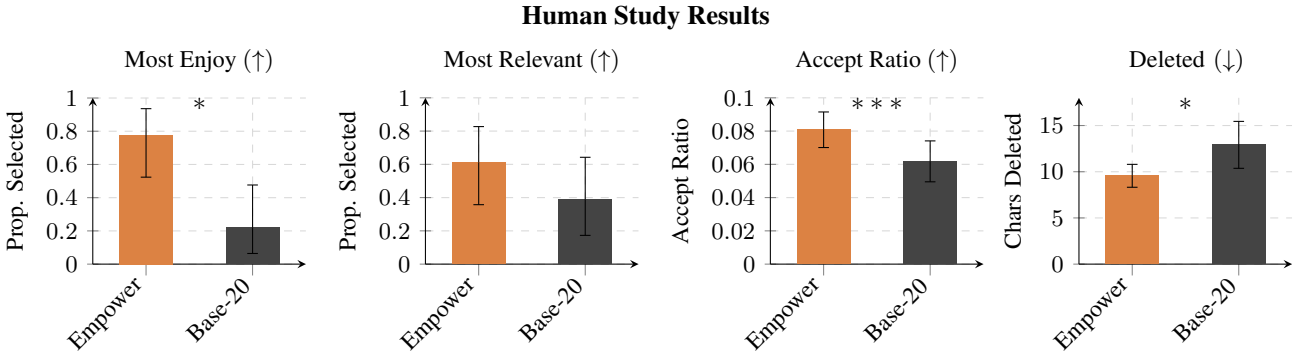


*Figure 3.* **Human study results with the Llama-3.1-8B-Instruct assistant**. Exact 95% confidence intervals are shown for Most Enjoy and Most Relevant as they represent Bernoulli data. Standard error bars are shown for Accept Ratio and Characters Deleted. In all cases, participants preferred using our `Empower` assistant.

**Quantitative Results.** Comparisons between the baselines and our method with $\eta = 0.32$ are shown in Fig. 2, using Gemma-3-27B-it as the simulated human model. `Empower` outperforms all baselines on pass@1, accept ratio, and DPR. It is worth noting that the accept ratios of `Empower` and Base-10 are close for Llama-3.1-8B-Instruct and Qwen3-8B . However, in those cases, acceptance ratio does not correspond to a higher Pass@1, and therefore the DPR is lower.

We also perform the same set of experiments with Llama-3.3-70B-Instruct as the human model, for which we show results in Table 2 of Section B. `Empower` similarly beats the baseline on pass@1 and DPR. See Section B for the full numeric results.

**Ablations.** We ablate the choice of $\eta$ and show results in Table 1. Our method outperforms the baselines in pass@1 and Discounted Pass Rate at all choices of $\eta$ we tried. The specific $\eta = 0.32$ we use in the core results was the first one we tried, as it corresponds to a cumulative likelihood threshold of 80%. The robustness of our method to the choice of $\eta$ and assistant model suggest that LLM calibration is not a significant issue.

We also investigate whether the stronger results are simply because the assistant stays passive. The Base-10 baseline, which caps the number of tokens per suggestion to 10, has an average suggestion length of 9.3 in the simulated results. $\eta = 0.5$ has an average suggestion length of 8.0 tokens, and $\eta = 1$ has an average suggestion length of 11.5 tokens. Table 1 shows that both $\eta = 0.5$ and $\eta = 1$ outperform Base-10 in Pass@1 and Discounted Pass Rate. This suggests that the primary gain of our method is not from giving shorter suggestions on average, but instead from intelligently knowing when to give a short suggestion or a longer suggestion, based on whether the assistant is confident in what the user would do.

We also include RFT results in Table 1. RFT has a higher acceptance ratio than the other SFT baselines, but fails to beat our offline method despite its use of user feedback. Finally, we evaluate GPT-5-mini on our simulated environment in Table 1 to compare our methods with a commercial system (Singh et al., 2025). As expected, GPT-5-mini outperforms all methods, although its acceptance ratio is not significantly higher than `Empower`'s.

### 5.3 Human Study: Evaluating Empowerment for Real-World Code Assistance

We also conducted an 18-person double-blinded user study in a code-generation setting with an assistant, similar to GitHub Copilot. The human study was conducted with Institutional Review Board (IRB) approval (Protocol #2025-03-18427). Participants were randomly assigned to complete one of two python coding problems with corresponding test cases. This editor is very similar to Nano, and, unlike in the simulated results, allows the user to edit previous code. They first spent 25 minutes attempting the problem with no assistant. Then, they spent 15 minutes attempting the problem with Assistant 1, took notes on what they liked and didn't like about it, and then repeated this step for Assistant 2. Finally, they were asked to rank the assistants on several metrics including how relevant they found the suggestions and which assistant they would most enjoy using in practice. The order of the assistants was randomized and hidden from the researcher's view.

To choose which two assistants to compare, we ran a pilot study with Llama-3.1-8B-Instruct as the assistant. Participants in the pilot tended to prefer `Empower` with $\eta = 4$, and the Base-20 baseline, so we chose these to focus on for the full study.

**Survey Results.** We show the results of the study in Fig. 3. Participants ranked the `Empower` assistant as the one they would more enjoy using in practice 78% of the

time ($p = 0.015$). Additionally, they ranked our assistant as providing more relevant suggestions 61% of the time, although the result was not statistically significant ($p = 0.240$). Although both assistants tended to provide relevant suggestions, the `Empower` assistant provided fewer suggestions overall. Participants preferred this approach to assistance, which we attribute to the empowerment objective teaching the model to only complete as long as it is confident about what the user will type next.

**Quantitative Results.** We also collected quantitative data about the user-assistant interaction. The `Empower` assistant had an acceptance rate of 8.08% compared to the 6.18% of the Base-20 assistant. Participants accepted suggestions from our assistant more ($p = 0.0002$). Participants also tended to delete more accepted text from the Base-20 assistant than from ours. The average number of deleted characters per accepted suggestion was 12.91 for Base-20 and 9.56 for ours ($p = 0.0118$). On average, `Empower` suggested 208 suggestions per user, whereas Base-20 suggested 333. The baseline also tended to give longer suggestions, at 82.2 characters per suggestion compared to 43.6 for `Empower`.

**Example Final Programs.** We include several example final programs in Section E.6 with both our `Empower` assistant and the Base-20 assistant. These demonstrate that, while `Empower` is often more conservative than the baseline, it still makes meaningful contributions to the final program.

## 6 Discussion

We show how assistive (LLM) agents can provide their own feedback signal for learning by estimating how empowered a human coder is. Our logit threshold method tractably computes empowering suggestions, which maximize the impact that the human will have.

While we demonstrated success in coding assistance, we expect that LLM assistants trained with empowerment can be useful in many domains, from general writing assistance to designing better interfaces for human-computer interaction. These also include more agentic applications where the assistant can infer when the human would predictably take an action, and instead take the action automatically. Our work enables the training of these agents at scale by simply configuring the likelihood estimator for a given domain.

**Limitations.** All experiments were conducted on competitive programming problems. Real-world code will often differ significantly in style and difficulty, which may require a more robust marginal likelihood estimator. The application of empowerment to more general coding tasks is left for future work.

Our empowerment estimator works well for a coding environment where the human has full control over the future state. The estimator may become less accurate in an environment where some actions lead to noisier transitions.

## Impact Statement

Empowerment methods may be used to create better assistive agents, improving the experience of people who collaborate with LLMs. There is a risk of an assistant being trained to self-empower, which would create a general power-seeking agent. However, our methods are focused on human-AI collaboration, which does not pose this risk.

## Acknowledgments

We would like to thank Cameron Allen, Jessy Lin, Michelle Li, Eli Bronstein, and Divy Thakkar for insightful discussions and feedback on earlier versions of this paper.

We would also like to thank Imbue AI for generously providing compute resources for experiments. This research was partly supported by National Science Foundation under Award No. 2441665, NSF IIS-2246811, the DoD NDSEG Fellowship, Schmidt Sciences, and Open Philanthropy. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

## References

Abel, D., Bowling, M., Barreto, A., Dabney, W., Dong, S., Hansen, S., Harutyunyan, A., Khetarpal, K., et al. Plasticity as the Mirror of Empowerment. arXiv:2505.10361, 2025.

Carroll, M., Foote, D., Siththaranjan, A., Russell, S., and Dragan, A. AI Alignment With Changing and Influenceable Reward Functions. *International Conference on Machine Learning*, 2024.

Carroll, M., Shah, R., Ho, M.K., Griffiths, T.L., Seshia, S.A., Abbeel, P., and Dragan, A. On the Utility of Learning About Humans for Human-AI Coordination. *Neural Information Processing Systems*, 2019.

Casper, S., Davies, X., Shi, C., Gilbert, T.K., Scheurer, J., Rando, J., Freedman, R., Korbak, T., et al. Open Problems and Fundamental Limitations of Reinforcement Learning From Human Feedback. arXiv:2307.15217, 2023.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.D.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating Large Language Models Trained on Code. arXiv:2107.03374, 2021.

Choi, J., Sharma, A., Lee, H., Levine, S., and Gu, S.S. Variational Empowerment as Representation Learning for Goal-Conditioned Reinforcement Learning. *International Conference on Machine Learning*, pp. 1953–1963, 2021.

Christiano, P., Leike, J., Brown, T.B., Martic, M., Legg, S., and Amodei, D. Deep Reinforcement Learning From Human Preferences. *Neural Information Processing Systems*, 2017.

de Abril, I.M. and Kanai, R. A Unified Strategy for Implementing Curiosity and Empowerment Driven Reinforcement Learning. arXiv:1806.06505, 2018.

DeepSeek-AI, Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., et al. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. arXiv:2501.12948, 2025.

Dong, H., Xiong, W., Goyal, D., Zhang, Y., Chow, W., Pan, R., Diao, S., Zhang, J., Shum, K., and Zhang, T. RAFT: Reward rAnked FineTuning for Generative Foundation Model Alignment. arXiv:2304.06767, 2023.

Du, Y., Tiomkin, S., Kiciman, E., Polani, D., Abbeel, P., and Dragan, A.D. AvE: Assistance via Empowerment. *CoRR*, 2020.

Eysenbach, B., Gupta, A., Ibarz, J., and Levine, S. Diversity is all you need: Learning skills without a reward function. *arXiv preprint arXiv:1802.06070*, 2018.

Giulianelli, M., Baan, J., Aziz, W., Fernández, R., and Plank, B. What Comes Next? Evaluating Uncertainty in Neural Text Generators Against Human Production Variability. H. Bouamor, J. Pino, and K. Bali, editors, *Empirical Methods in Natural Language Processing*, pp. 14349–14371, 2023.

Gopnik, A. Empowerment as Causal Learning, Causal Learning as Empowerment: A Bridge Between Bayesian Causal Hypothesis Testing and Reinforcement Learning. 2024.

Grattafiori, A., Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., et al. The Llama 3 Herd of Models. arXiv:2407.21783, 2024.

Hadfield-Menell, D., Milli, S., Abbeel, P., Russell, S.J., and Dragan, A. Inverse Reward Design. *Neural Information Processing Systems*, 30, 2017.

Hadfield-Menell, D., Russell, S.J., Abbeel, P., and Dragan, A. Cooperative Inverse Reinforcement Learning. *Neural Information Processing Systems*, 29, 2016.

Hejna, J. and Sadigh, D. Inverse Preference Learning: Preference-Based RL Without a Reward Function. *Neural Information Processing Systems*, 2023.

Huang, A., Block, A., Foster, D.J., Rohatgi, D., Zhang, C., Simchowitz, M., Ash, J.T., and Krishnamurthy, A. Self-Improvement in Language Models: The Sharpening Mechanism. *International Conference on Learning Representations*, 2025.

Huang, J., Gu, S., Hou, L., Wu, Y., Wang, X., Yu, H., and Han, J. Large Language Models Can Self-Improve. *Empirical Methods in Natural Language Processing*, pp. 1051–1068, 2023.

Jain, N., Han, K., Gu, A., Li, W.D., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., and Stoica, I. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. arXiv:2403.07974, 2024.

Klyubin, A.S., Polani, D., and Nehaniv, C.L. Empowerment: A Universal Agent-Centric Measure of Control. *IEEE Congress on Evolutionary Computation*, volume 1, pp. 128–135, 2005.

Klyubin, A.S., Polani, D., and Nehaniv, C.L. Keep Your Options Open: An Information-Based Driving Principle for Sensorimotor Systems. *PLOS One*, 3(12):e4018, 2008.

Laidlaw, C., Bronstein, E., Guo, T., Feng, D., Berglund, L., Svegliato, J., Russell, S., and Dragan, A. Scalably Solving Assistance Games. *ICML 2024 Workshop on Models of Human Feedback for AI Alignment*, 2024.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., et al. Self-Refine: Iterative Refinement With Self-Feedback. *Neural Information Processing Systems*, 36, 2024.

Mohamed, S. and Jimenez Rezende, D. Variational information maximisation for intrinsically motivated reinforcement learning. *Advances in neural information processing systems*, 28, 2015.

Mullen, Jr., J.F. and Manocha, D. LBAP: Improved Uncertainty Alignment of LLM Planners Using Bayesian Inference. arXiv:2403.13198, 2024.

Myers, V., Ellis, E., Levine, S., Eysenbach, B., and Dragan, A. Learning to Assist Humans Without Inferring Rewards. *Neural Information Processing Systems*, 2024.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C.L., Mishkin, P., Zhang, C., Agarwal, S., et al. Training Language Models to Follow Instructions With Human Feedback. arXiv:2203.02155, 2022.

Pang, J.C., Wang, P., Li, K., Chen, X.H., Xu, J., Zhang, Z., and Yu, Y. Language Model Self-Improvement by Reinforcement Learning Contemplation. arXiv:2305.14483, 2023.

Rafailov, R., Sharma, A., Mitchell, E., Ermon, S., Manning, C.D., and Finn, C. Direct Preference Optimization: Your Language Model Is Secretly a Reward Model. *Neural Information Processing Systems*, 2024.

Ramrakhya, R., Chang, M., Puig, X., Desai, R., Kira, Z., and Mottaghi, R. Grounding Multimodal LLMs to Embodied Agents That Ask for Help With Reinforcement Learning. arXiv:2504.00907, 2025.

Ren, A.Z., Dixit, A., Bodrova, A., Singh, S., Tu, S., Brown, N., Xu, P., Takayama, L., Xia, F., Varley, J., et al. Robots That Ask for Help: Uncertainty Alignment for Large Language Model Planners. *Conference on Robot Learning*, pp. 661–682, 2023.

Salge, C., Glackin, C., and Polani, D. Empowerment–an Introduction. *Guided Self-Organization: Inception*, (arXiv:1310.1863):67–114, 2014.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal Policy Optimization Algorithms. *CoRR*, 2017.

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language Agents With Verbal Reinforcement Learning. *Neural Information Processing Systems*, 36, 2024.

Singh, A., Fry, A., Perelman, A., Tart, A., Ganesh, A., El-Kishky, A., McLaughlin, A., Low, A., et al. OpenAI GPT-5 System Card. arXiv:2601.03267, 2025.

Song, J., Gore, J., and Kleiman-Weiner, M. Estimating the Empowerment of Language Model Agents. arXiv:2509.22504, 2025.

Stiennon, N., Ouyang, L., Wu, J., Ziegler, D.M., Lowe, R., Voss, C., Radford, A., Amodei, D., and Christiano, P.F. Learning to Summarize From Human Feedback. *CoRR*, 2020.

Sutton, R.S. and Barto, A.G. *Reinforcement Learning: An Introduction*. MIT press, 2018.

Team, G., Kamath, A., Ferret, J., Pathak, S., Vieillard, N., Merhej, R., Perrin, S., Matejovicova, T., et al. Gemma 3 Technical Report. arXiv:2503.19786, 2025.

Trivedi, H., Khot, T., Hartmann, M., Manku, R., Dong, V., Li, E., Gupta, S., Sabharwal, A., and Balasubramanian, N. AppWorld: A Controllable World of Apps and People for Benchmarking Interactive Coding Agents. *Annual Meeting of the Association for Computational Linguistics*, pp. 16022–16076, 2024.

Williams, M., Carroll, M., Narang, A., Weisser, C., Murphy, B., and Dragan, A. On Targeted Manipulation and Deception When Optimizing LLMs for User Feedback. *International Conference on Learning Representations*, 2024.

Xie, L., Liu, H., Zeng, J., Tang, X., Han, Y., Luo, C., Huang, J., Li, Z., Wang, S., and He, Q. A Survey of Calibration Process for Black-Box LLMs. arXiv:2412.12767, 2024.

Yang, A., Li, A., Yang, B., Zhang, B., Hui, B., Zheng, B., Yu, B., Gao, C., et al. Qwen3 Technical Report. arXiv:2505.09388, 2025.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T., Cao, Y., and Narasimhan, K. Tree of Thoughts: Deliberate Problem Solving With Large Language Models. *Neural Information Processing Systems*, 36, 2024.

Yuan, W., Pang, R.Y., Cho, K., Li, X., Sukhbaatar, S., Xu, J., and Weston, J.E. Self-Rewarding Language Models. *International Conference on Machine Learning*, 2024.

Yuan, Z., Yuan, H., Li, C., Dong, G., Lu, K., Tan, C., Zhou, C., and Zhou, J. Scaling Relationship on Learning Mathematical Reasoning With Large Language Models. arXiv:2308.01825, 2023.

# A Website and Code

The website that links to the code and configs to reproduce our experiments can be found at https://empowerment-for-llms.github.io/.

# B Additional Simulated Results

Full experimental results are presented in Tables 2 and 3. We ablated the choice of human model, also training models on a Llama-3.3-70B-Instruct generated dataset and using it as the simulated human.

# C Training Details

Experiments were performed on a NVIDIA H100 node with 8 GPUs, each with 80GB of VRAM. Pre-trained weights were taken from the LLaMA-3.1-8B, LLaMA-3.3-70B (Grattafiori et al., 2024), Qwen3-8B , Qwen3-14B (Yang et al., 2025), and Gemma-3-27B-it (Team et al., 2025) models, as described in Section 5. We finetuned the assistant for one epoch on a dataset of 4,138 examples with a test split size of 0.2.

The LlaMA models were used under the Llama 3.1 Community License Agreement. The Qwen models were used under the Apache 2.0 license. Gemma was used under the Gemma Terms of Use. Our training dataset was initialized from MatrixStudio/Codeforces-Python-Submissions, https://huggingface.co/datasets/MatrixStudio/Codeforces-Python-Submissions. We de-duplicated the problems and re-generated the solutions using the corresponding human model that was being assisted.

# D Prompts

We provide prompts used for the LLMs in our experiments.

## D.1 Assistant Prompt

The assistant system prompt is:

```
You are assisting a human in a python code generation task. Your role is to provide suggested completions given
what they have already typed. Please try to infer what the human wants the next piece of code to be given the
code they have already written. If they have not written any code, please provide a good start to their program, such as with import statements or function definitions.

The way you will compose your suggestion is by providing the next version of the code which would replace the current code.
Please re-type the current code and then add in your suggested completion.
DO NOT output any other text, including no quotation marks.

## Remember to always re-type the code written so far and then add in your suggested completion.
If you don't re-type the code written so far *exactly as it is written* (with all of the functions, comments, import statements, etc)
an error will be raised.
```

The assistant user prompt is:

```
Now it's your turn! Please provide a completion for the following code:
```python
{{ code_to_complete }}
```
```

## D.2 Few Shot Examples for Assistant

```
# > user
```

*Table 2.* Assistant results with Llama-3.3-70B-Instruct as the human model

| Base Model | Name | Pass@1 (↑) | Accept Ratio (↑) | Discounted Pass Rate (↑) |
|---|---|---|---|---|
| Qwen3-8B | Empower | **0.218** $^{(\pm 0.019)}$ | 0.488 $^{(\pm 0.024)}$ | **0.176** $^{(\pm 0.016)}$ |
| Qwen3-8B | SFT-20 | 0.167 $^{(\pm 0.018)}$ | 0.192 $^{(\pm 0.018)}$ | 0.116 $^{(\pm 0.013)}$ |
| Qwen3-8B | SFT-10 | 0.152 $^{(\pm 0.017)}$ | 0.299 $^{(\pm 0.021)}$ | 0.114 $^{(\pm 0.013)}$ |
| Qwen3-8B | SFT-RAND-1-30 | 0.156 $^{(\pm 0.017)}$ | 0.201 $^{(\pm 0.019)}$ | 0.109 $^{(\pm 0.012)}$ |
| Qwen3-8B | Base-10 | 0.198 $^{(\pm 0.019)}$ | **0.592** $^{(\pm 0.023)}$ | 0.162 $^{(\pm 0.015)}$ |
| Qwen3-8B | Base | 0.183 $^{(\pm 0.018)}$ | 0.351 $^{(\pm 0.022)}$ | 0.143 $^{(\pm 0.014)}$ |
| Llama3.1-8B Instruct | Empower | **0.282** $^{(\pm 0.021)}$ | 0.317 $^{(\pm 0.022)}$ | **0.208** $^{(\pm 0.016)}$ |
| Llama3.1-8B Instruct | SFT-20 | 0.097 $^{(\pm 0.014)}$ | 0.165 $^{(\pm 0.017)}$ | 0.066 $^{(\pm 0.010)}$ |
| Llama3.1-8B Instruct | SFT-10 | 0.104 $^{(\pm 0.014)}$ | 0.257 $^{(\pm 0.021)}$ | 0.074 $^{(\pm 0.010)}$ |
| Llama3.1-8B Instruct | SFT-RAND-1-30 | 0.112 $^{(\pm 0.015)}$ | 0.184 $^{(\pm 0.018)}$ | 0.075 $^{(\pm 0.010)}$ |
| Llama3.1-8B Instruct | Base-10 | 0.156 $^{(\pm 0.017)}$ | **0.537** $^{(\pm 0.023)}$ | 0.127 $^{(\pm 0.014)}$ |
| Llama3.1-8B Instruct | Base | 0.170 $^{(\pm 0.018)}$ | 0.297 $^{(\pm 0.021)}$ | 0.134 $^{(\pm 0.014)}$ |
| Qwen3-14B | Empower | **0.249** $^{(\pm 0.020)}$ | 0.459 $^{(\pm 0.023)}$ | **0.201** $^{(\pm 0.016)}$ |
| Qwen3-14B | SFT-20 | 0.145 $^{(\pm 0.017)}$ | 0.188 $^{(\pm 0.018)}$ | 0.102 $^{(\pm 0.012)}$ |
| Qwen3-14B | SFT-10 | 0.165 $^{(\pm 0.017)}$ | 0.292 $^{(\pm 0.021)}$ | 0.126 $^{(\pm 0.013)}$ |
| Qwen3-14B | SFT-RAND-1-30 | 0.145 $^{(\pm 0.017)}$ | 0.226 $^{(\pm 0.020)}$ | 0.106 $^{(\pm 0.012)}$ |
| Qwen3-14B | Base-10 | 0.174 $^{(\pm 0.018)}$ | **0.597** $^{(\pm 0.023)}$ | 0.143 $^{(\pm 0.015)}$ |
| Qwen3-14B | Base | 0.161 $^{(\pm 0.017)}$ | 0.299 $^{(\pm 0.021)}$ | 0.127 $^{(\pm 0.014)}$ |

We evaluate on 554 LiveCodeBench problems, and find that `Empower` outperforms all baselines in terms of Pass@1 and DPR. Standard errors are shown in parentheses.

*Table 3.* **Assistant results with Gemma-3-27B-it as the human model.** We evaluate on 554 LiveCodeBench problems. We find `Empower` outperforms all baselines in terms of Pass@1 and DPR. Standard errors are shown in parentheses.

| Base Model | Name | Pass@1 (↑) | Accept Ratio (↑) | Discounted Pass Rate (↑) |
|---|---|---|---|---|
| Qwen3-8B | Empower | **0.178** $^{(\pm 0.018)}$ | **0.630** $^{(\pm 0.023)}$ | **0.156** $^{(\pm 0.016)}$ |
| Qwen3-8B | SFT-20 | 0.086 $^{(\pm 0.013)}$ | 0.299 $^{(\pm 0.021)}$ | 0.072 $^{(\pm 0.011)}$ |
| Qwen3-8B | SFT-10 | 0.101 $^{(\pm 0.014)}$ | 0.367 $^{(\pm 0.023)}$ | 0.086 $^{(\pm 0.012)}$ |
| Qwen3-8B | Base-10 | 0.090 $^{(\pm 0.013)}$ | 0.582 $^{(\pm 0.023)}$ | 0.080 $^{(\pm 0.012)}$ |
| Qwen3-8B | Base | 0.092 $^{(\pm 0.014)}$ | 0.400 $^{(\pm 0.023)}$ | 0.083 $^{(\pm 0.012)}$ |
| Llama3.1-8B Instruct | Empower | **0.176** $^{(\pm 0.018)}$ | **0.670** $^{(\pm 0.022)}$ | **0.150** $^{(\pm 0.015)}$ |
| Llama3.1-8B Instruct | SFT-20 | 0.070 $^{(\pm 0.012)}$ | 0.231 $^{(\pm 0.020)}$ | 0.057 $^{(\pm 0.010)}$ |
| Llama3.1-8B Instruct | SFT-10 | 0.062 $^{(\pm 0.011)}$ | 0.268 $^{(\pm 0.021)}$ | 0.053 $^{(\pm 0.010)}$ |
| Llama3.1-8B Instruct | Base-10 | 0.064 $^{(\pm 0.011)}$ | 0.649 $^{(\pm 0.022)}$ | 0.057 $^{(\pm 0.010)}$ |
| Llama3.1-8B Instruct | Base | 0.064 $^{(\pm 0.011)}$ | 0.383 $^{(\pm 0.023)}$ | 0.055 $^{(\pm 0.010)}$ |
| Qwen3-14B | Empower | **0.170** $^{(\pm 0.018)}$ | **0.659** $^{(\pm 0.022)}$ | **0.148** $^{(\pm 0.015)}$ |
| Qwen3-14B | SFT-20 | 0.088 $^{(\pm 0.013)}$ | 0.381 $^{(\pm 0.023)}$ | 0.077 $^{(\pm 0.012)}$ |
| Qwen3-14B | SFT-10 | 0.101 $^{(\pm 0.014)}$ | 0.461 $^{(\pm 0.023)}$ | 0.088 $^{(\pm 0.012)}$ |
| Qwen3-14B | Base-10 | 0.062 $^{(\pm 0.011)}$ | 0.530 $^{(\pm 0.023)}$ | 0.055 $^{(\pm 0.010)}$ |
| Qwen3-14B | Base | 0.086 $^{(\pm 0.013)}$ | 0.312 $^{(\pm 0.022)}$ | 0.077 $^{(\pm 0.012)}$ |

```
# > Now it's your turn! Please provide a completion for the following code:
# ```python
def twoSum(self, nums: List[int], target: int) -> List[int]:
    numMap = {}
    n = len(nums)

    # Build the hash table
    for i in range(n):
        numMap[nums[i]] = i

    # Find the complement
    for i in range(n):

# ```
# > assistant
# > Here is my suggested completion:
# ```python
def twoSum(self, nums: List[int], target: int) -> List[int]:
    numMap = {}
    n = len(nums)

    # Build the hash table
    for i in range(n):
        numMap[nums[i]] = i

    # Find the complement
    for i in range(n):
        complement = target - nums[i]
# ```
# > user
# > Now it's your turn! Please provide a completion for the following code:
# ```python
def whoami(name:
# ```
# > assistant
# > Here is my suggested completion:
# ```python
def whoami(name: str, age: int) -> str:
# ```
```

### D.3 Human Model Prompts

**Human Appender Prompt.** The prompts given to the human when they are deciding what to write next. They are provided both a system prompt and a user prompt. The system prompt is the following:

```
You are an expert Python programmer. You will be given a question (problem
    specification) and will generate a correct Python program that matches the
    specification and passes all tests.
*Please do not provide any sample outputs or testcases in your response.
    Additionally, you are only allowed to solve the problem *ONCE*.
Do not attempt to retry your solution if you are unhappy with it.
For example, if your solution is in a function called `solve`, you should only
    define one function called `solve`. DO NOT try to retry it if you think it has a
     bug.
For example, you should not write `solve2` if you think `solve` has a bug. Only the
     first solution will be counted, so simply stop writing once the first solution
    is finished--even if it is not correct.
```

The user prompt is the following:

```
### Question:
{{problem.question_content}}

### Format:
{
```

```
You will use the following starter code to write the solution to the problem and enclose your code within delimiters.
```python
{{ problem.starter_code }}
```
{
Read the inputs from stdin solve the problem and write the answer to stdout (do not directly test on the sample inputs). Enclose your code within delimiters as follows. Ensure that when the python program runs, it reads the inputs, runs the algorithm and writes output to STDOUT.
```python
# YOUR CODE HERE
```
{
```

**Human Acceptor Prompt.** When the human is choosing whether or not to accept or reject a suggestion, it has a different prompt. There is a system prompt, a user prompt that encourages it to reason about the suggestion, and finally a user prompt that asks it for its final decision to accept, reject, or finish. The system prompt is the following:

```
You are an expert Python programming agent. You will be given a question (problem specification) and will generate a correct Python program that matches the specification and passes all tests.

The way you will compose your program is by interacting with the file editing interface through tools.
In addition, you will be assisted by an automated assistant that will sometimes propose suggested completions to whatever you are typing.
At every step you will first write your thoughts on the suggestion and whether it correctly continues your solution, or instead introduces bugs.
Please *only consider the assistant suggestion* in your decision to accept the suggestion, reject the suggestion, or finish and submit your code for testing.
Previous code that you have written is not able to be changed, so you should only look at the changes the assistant is proposing.
At the very end of your thought process, you will write one word to specify which action you are taking.
Unless you take the "finish" action, you will be prompted afterwards to write what you would like to append to your program.
Therefore, please accept suggestions as long as they do not introduce bugs, and either help you solve the problem or improve the quality of your code.
It's ok if the the suggested completion is incomplete, because you can always append to it later.

You will first be prompted to write your thoughts on the suggestion.
Afterwords, you will be prompted to write which action you would like to take.

Here are the actions you have access to:
### "accept"
    description: Accepts a suggested completion given by an intelligent assistant. The suggested completion will then be incorporated into the code you have written.
### "reject"
    description: Rejects a suggested completion given by an intelligent assistant. The suggested completion will not be incorporated into the code you have written.
### "finish"
    description: Tells the editor that you have finished writing the program and to run the testcases. Only call this tool if you are confident that your program is finished. You will not be prompted to write any more code after calling this tool.

Remember, you will be able to continue writing your program regardless of whether you accept the suggested completion or not.
As long as the suggested completion does not introduce bugs, and either helps you solve the problem or improves the quality of your code, you should accept it.
```

```
DO NOT reject a suggestion because it is "minor" or "short". Only reject the suggestion if it is wrong, introduces a bug, or otherwise sets you back.
```

The user reasoning prompt is the following:

```
## You have written the following code:
```python
{code}
```

## Suggested Completion
Here is what your code would look like with a suggested completion:
```python
{suggestion}
```

## Suggested Completion diff
For clarity, here is the diff between your current code and the suggested completion code:
{git_diff_string(code, suggestion)}

## Instructions:
What do you think of the suggested completion? Do you think it is solving the question correctly, or does it introduce a bug or error?
Please write down your thoughts. You are not allowed to write any new code in your response, only your thoughts on whether the suggested completion helps you on your way to solving the problem, or otherwise improves the quality of your code.
It is ok if the the suggested completion is incomplete, because you will be prompted to append to it later.
You are also not able to take any actions at this stage.
```

After it has provided reasoning for whether or not it believes the suggestion is a good one, we prompt it to make its final decision with the following user prompt:

```
Now, please write which action you would like to take.
Remember, the actions available are "accept" to accept the suggested completion, "reject" to reject the suggested completion, and "finish" to finish writing your code and run the tests.
Please only call "finish" if you are confident that your code is correct and you are ready to run the tests.
```

# E Human Study

## E.1 Study Instructions

SETUP

1. Sign research consent form.
2. Run locally: `git clone https://github.com/festusev/tab_frontend`
3. If on Mac:
   3.1. Navigate to the cloned repository and run: `./install.sh`
4. If on Windows:
   4.1. Install node: https://nodejs.org/en/download
   4.2. Run `npm install`
   4.3. Run `npm start`
5. If on Linux:
   5.1. Run `sudo apt install nodejs npm`
6. Enter your name in the box.

7. Switch the assistant to **Assistant 1**. Open up the scratchpad, type a few things, and make sure that a suggestion appears (suggestions will not always appear).
8. You can accept suggestions using the `Tab` key.
9. You can explicitly reject a suggestion using the `Esc` key.
10. Click **Back to Launch** at the top of the window.
11. Switch the assistant to **No Assistant**.
12. Move on to the Study section.

STUDY

1. You are only allowed to use the Python docs: https://docs.python.org/3/. You may not use anything else on the internet.
2. To run the test cases:
   2.1. Save your file (`CMD + S`).
   2.2. At the top of the editor, click **Run Testcases**. This will copy a command to your clipboard which you can then paste and run in your terminal.
3. Set a timer for 25 minutes.
4. Switch the assistant to **No Assistant**.
5. Begin the problem.
6. Whenever you or the timer finish, switch to **Assistant 1**.
7. Open up the same problem.
8. Set a timer for 15 minutes and solve the problem with Assistant 1. Pay attention to what you like and dislike about this assistant.
9. Fill out your notes in this form: redacted.
10. Save the file you are working on (`CMD + S`).
11. Repeat steps 6–10 for Assistant 2.
12. Complete the rest of the form and rank the assistants.
13. Make sure to zip and upload your `problems` directory to the form.

### E.2 Problem 1: Lava Trap

Simulate a single player walking on a square grid with lava squares. After each command, print if they fell into the lava, or, if they survived, print the player's current row, column, and facing.

The player will never move out of bounds of the grid. The top left of the grid is $(1, 1)$ and the bottom right is $(N, N)$.

BOARD

- An $N \times N$ grid of characters:
  - `.` — empty cell
  - `L` — lava
- Cells are 1-indexed: row $1..N$, column $1..N$.

PLAYER

- Starts at row $r$, column $c$, facing $dir \in \{\texttt{U}, \texttt{D}, \texttt{L}, \texttt{R}\}$.

COMMANDS

You are given $Q$ commands:

1. `MOVE`
   Move forward one step in the current direction.
2. `FACE X` where $X \in \{\texttt{U}, \texttt{D}, \texttt{L}, \texttt{R}\}$
   Set the facing direction.

TILE EFFECTS (AFTER THE MOVE)

- If the player moves into lava, the simulation ends, and `Game Over` is printed.

INPUT

The input will come from standard in:

1. $N$
2. $N$ lines of grid (each of length $N$)
3. $r\ c\ dir$
4. $Q$
5. $Q$ lines of commands

CONSTRAINTS

- $2 \leq N \leq 50$
- $1 \leq r, c \leq N$
- $dir \in \{\texttt{U}, \texttt{D}, \texttt{L}, \texttt{R}\}$
- Commands:
  - `MOVE`
  - `FACE U|D|L|R`
- $1 \leq Q \leq 2 \times 10^5$

OUTPUT

After each command, print one line:

```
r c dir
```

(with the player's 1-indexed row/col and facing as `U|D|L|R`).

EXAMPLE

**Input:**

```
3
..L
...
...
2 2 R
3
MOVE
FACE U
MOVE
```

**Output:**

```
2 3 R
2 3 U
Game Over
```

STARTER CODE

```python
import sys


def read_grid():
    """Reads N and then N lines of the grid. Returns (N, grid)."""
    n = int(sys.stdin.readline().strip())
    grid = [list(sys.stdin.readline().strip()) for _ in range(n)]
    return n, grid


def read_starting_position():
    """Reads r, c, dir. Returns (r, c, dir)."""
    parts = sys.stdin.readline().split()
    r, c, d = int(parts[0]), int(parts[1]), parts[2]
    return r, c, d

def read_q():
    """Reads q from stdin."""
    return int(sys.stdin.readline().strip())

def read_next_move():
    """Reads and returns the next command as a string, or None if EOF."""
    line = sys.stdin.readline()
    if not line:
        return None
    return line.strip()


def main():
```

### E.3 Problem 2: Special Keyboard

Simulate a user typing on a special keyboard. They will type one character at a time. After they have finished typing, print what they wrote.

INPUT

The input will come from standard in:

1. The number of characters that the user will type, $q$ ($1 \leq q \leq 2000$).
2. One character that the user types per line.
3. Characters may include letters, digits, spaces, punctuation, and the markers below.

OUTPUT

- One line: the transformed string.

SPECIAL TOGGLES

Most keyboards have a Caps Lock key that toggles between lowercase and uppercase letters. This special keyboard has that, in addition to several non-standard toggles. When the user types a special toggle key, turn the toggle **on**, and apply its rule for all of the text that the user types until they type the special toggle key again to turn it **off**.

- Toggle keys do not affect previously written text, only future text.
- Do not append the toggle character to the user's output.

- More than one toggle may be active at the same time.

TOGGLE RULES

- `^` → Caps Lock: uppercase all letters while this toggle is active. (In Python: `s.upper()`)
- `~` → While active, consonants (letters that are not vowels) are duplicated, preserving case. ("y" counts as a consonant.)
- `#` → While active, only digits and the *first* "`.`" encountered are appended to the output.
  - Skip all other characters.
  - If a second "`.`" appears (or any additional one), skip it.

  (In Python: check if a character is a digit with `s.isdigit()`.)

EXAMPLES

**Input**

```
5
^
a
b
^
c
```

**Output**

```
ABc
```

**Input**

```
9
^
a
~
b
c
~
d
^
e
```

**Output**

```
ABBCCDe
```

**Input**

```
18
^
d
~
b
#
I
```

```
6
7
.
9
.
1
Z
#
a
^
c
.
```

**Output**

```
DBB67.91Acc.
```

STARTER CODE

```python
import sys
from typing import List, Tuple


def read_q() -> int:
    """Read the number of typed characters (q) from the first line."""
    line = sys.stdin.readline()
    if not line:
        raise EOFError("Expected an integer q on the first line.")
    return int(line.strip())


def read_next_char() -> str:
    """
    Read the next 'character per line'.
    """
    line = sys.stdin.readline()
    if line == "":
        raise EOFError("Unexpected end of input while reading characters.")
    # Take the first character on the line.
    return line[0]


def main() -> None:
```

### E.4 Questionnaire

This questionnaire was given to participants through a Google Form.

1. What is your name?
2. Which question are you solving?
3. Assistant 1 Notes (Paragraph entry).
4. Assistant 2 Notes (Paragraph entry).
5. How relevant are the assistant's suggestions? (Assign each label to only one assistant)
   (a) Assistant 1. [1 (Most relevant suggestions) or 2 (Least relevant suggestions)]
   (b) Assistant 2. [1 (Most relevant suggestions) or 2 (Least relevant suggestions)]
6. How often did you have to delete the assistant's work? (Assign each label to only one assistant)
   (a) Assistant 1. [1 (Fewest deletes) or 2 (Most deletes)]
   (b) Assistant 2. [1 (Fewest deletes) or 2 (Most deletes)]

7. Which would you most enjoy using in practice? (Assign each label to only one assistant)
   (a) Assistant 1. [1 (Most enjoy) or 2 (Least enjoy)]
   (b) Assistant 2. [1 (Most enjoy) or 2 (Least enjoy)]

### E.5 Additional Results

In the survey we asked participants to rank the assistants based on how often they had to delete the assistant's work. In total, 17 out of the 18 participants ranked our `Empower` method over the baseline ($p = 0.00007$). As we also collected the participant's keypresses, we instead included the exact number of characters which were accepted and later deleted in the main text, which is a more informative metric.

### E.6 Example Final Programs

We include examples of final programs written by two randomly chosen human study participants. The code is highlighted to indicate the source of each character: the starter code is colored beige, the participant code is colored blue, and the assistant code is colored green.

#### E.6.1 Lava Trap Base-20

The final program written by a participant with the Base-20 assistant on the Lava Trap problem.

USER ASSISTANT STARTER_CODE

```
import sys


def read_grid():
    """Reads N and then N lines of the grid. Re
turns (N, grid)."""
    n = int(sys.stdin.readline().strip())
    grid = [list(sys.stdin.readline().strip())
for _ in range(n)]
    return n, grid


def read_starting_position():
    """Reads r, c, dir. Returns (r, c, dir)."""
    parts = sys.stdin.readline().split()
    r, c, d = int(parts[0]), int(parts[1]), par
ts[2]
    return r, c, d

def read_q():
    """Reads q from stdin."""
    return int(sys.stdin.readline().strip())

def read_next_move():
    """Reads and returns the next command as a
string, or None if EOF."""
    line = sys.stdin.readline()
    if not line:
        return None
    return line.strip()


def main():
    q = read_q()
    n, grid = read_grid()
    r,c,d = read_starting_position()
    for _ in range(q):
        print(f"this is q{q}")
        next_move = read_next_move()
```

```
        print("running this here")
        # Process the move
        # For example, let's assume we want to
print the positi
        # Process the move here, for example:
        if next_move == "MOVE":
            # Get the direction
            print("running this")
            # Move in the given direction
            if d == "U":
                r -= 1
            elif d == "D":
                r += 1
            elif d == "L":
                c -= 1
            elif d == "R":
                c += 1
            if grid[r-1][c-1] == "L":
                print("Game Over")
                return
            else:
                print(f"{r} {c} {d}")
        else:
            d = next_move[-1]
            print(f"{r} {c} {d}")
```

### E.6.2 Lava Trap Empower

The final program written by the same participant with the `Empower` assistant on the Lava Trap problem.

USER ASSISTANT STARTER_CODE

```
import sys


def read_grid():
    """Reads N and then N lines of the grid. Re
turns (N, grid)."""
    n = int(sys.stdin.readline().strip())
    grid = [list(sys.stdin.readline().strip())
for _ in range(n)]
    return n, grid


def read_starting_position():
    """Reads r, c, dir. Returns (r, c, dir)."""
    parts = sys.stdin.readline().split()
    r, c, d = int(parts[0]), int(parts[1]), par
ts[2]
    return r, c, d

def read_q():
    """Reads q from stdin."""
    return int(sys.stdin.readline().strip())

def read_next_move():
    """Reads and returns the next command as a
string, or None if EOF."""
    line = sys.stdin.readline()
    if not line:
        return None
    return line.strip()


def main():
```

```
    n, grid = read_grid()
    r, c, dir = read_starting_position()
    q= read_q()


    while q >=0:
        next_dirs = {"L":-1, "R": 1, "U": -1, "
D": 1}
        m = read_next_move()
        if not m:
            return
        if m == "MOVE":
            if dir in ["U", "D"]:
                r += next_dirs[dir]
            else:
                c += next_dirs[dir]
            if grid[r-1][c-1] == "L":
                print("Game Over")
                return
            else:
                print(f"{r} {c} {dir}")
        else:
            new_dir = m[-1]
            dir = new_dir
            print(f"{r} {c} {dir}")
        q -= 1

main()
```

### E.6.3 Special Keyboard Base-20

The final program written by a different participant with the Base-20 assistant on the Special Keyboard problem.

USER ASSISTANT STARTER_CODE

```
import sys
from typing import List, Tuple


def read_q() -> int:
    """Read the number of typed characters (q)
from the first line."""
    line = sys.stdin.readline()
    if not line:
        raise EOFError("Expected an integer q o
n the first line.")
    return int(line.strip())


def read_next_char() -> str:
    """
    Read the next 'character per line'.
    """
    line = sys.stdin.readline()
    if line == "":
        raise EOFError("Unexpected end of input
 while reading characters.")
    # Take the first character on the line.
    return line[0]


def main() -> None:

    output = ''
```

```


    # Read the number of characters
    q = read_q()

    caps_flag = False
    duplicate_flag = False
    digits_flag = False
    saw_dot = False

    # Read and process
    for _ in range(q):
        char = read_next_char()

        if char == '^':
            caps_flag = not caps_flag
            continue
        if char == '#':
            digits_flag = not digits_flag
            saw_dot = False
            continue
        if char == '~':
            duplicate_flag = not duplicate_flag
            continue

        # Check if we need to process the chara
cter based on the flags
        if caps_flag:
            char = char.upper()
        if duplicate_flag:
            if char not in {'a', 'e', 'i', 'o',
 'u'}:
                char = 2*char
        if digits_flag:
            if char == '.' and not saw_dot:
                saw_dot = True
            elif not char.isdigit():
                char = ''

        output += char

    print(output)

main()

```

### E.6.4 Special Keyboard Empower

The final program written by the different participant with the `Empower` assistant on the Special Keyboard problem.

USER ASSISTANT STARTER_CODE

```
import sys
from typing import List, Tuple


def read_q() -> int:
    """Read the number of typed characters (q)
from the first line."""
    line = sys.stdin.readline()
    if not line:
        raise EOFError("Expected an integer q o
n the first line.")
    return int(line.strip())

```

```python

def read_next_char() -> str:
    """
    Read the next 'character per line'.
    """
    line = sys.stdin.readline()
    if line == "":
        raise EOFError("Unexpected end of input
 while reading characters.")
    # Take the first character on the line.
    return line[0]


def main():

    output = ''
    caps_flag = False
    digits_flag = False
    duplicate_flag = False
    saw_dot = False

    q = read_q()

    for _ in range(q):
        char = read_next_char()

        if char == '^': caps_flag = 1 - caps_fl
ag
        elif char == '~': duplicate_flag = 1 -
duplicate_flag
        elif char == '#': digits_flag = 1 - dig
its_flag

        if digits_flag:
            if char == '.' and saw_dot = True:
                char = ''
            if char == '.' and saw_dot = False:
                saw_dot = True

            elif not char.isdigit():
                char = ''

        elif caps_flag:
            char = char.upper()
        elif duplicate_flag:
            if char not in {'a', 'e', 'i', 'o',
 'u'}:
                char = char*2

        output += char

    return output


print(main())
```

## F LLM Acknowledgment

We did not use LLMs significantly in the writing or ideation of this paper. An LLM was used to proof-read, and a few sentences were reworded accordingly.