NNetNav: Unsupervised Learning of Browser Agents Through Environment Interaction in the Wild

Anonymous authors
Paper under double-blind review

Abstract

We introduce NNetNav, a method for unsupervised interaction with websites that generates synthetic demonstrations for training browser agents. Given any website, NNetNav produces these demonstrations by retroactively labeling action sequences from an exploration policy. Most work on training browser agents has relied on expensive human supervision, and the limited prior work on such interaction-based techniques has failed to provide effective search through the exponentially large space of exploration. In contrast, NNetNav exploits the hierarchical structure of language instructions to make this search more tractable: Complex instructions are typically decomposable into simpler sub-tasks, allowing NNetNav to automatically prune interaction episodes when an intermediate trajectory cannot be annotated with a meaningful sub-task. LLama-3.1-8b finetuned on 10k NNetNav self-generated demonstrations obtains over 16% success rate on WebArena, and 35% on WebVoyager, an improvement of 15pts and 31pts respectively over zero-shot LLama-3.1-8b, outperforming zero-shot GPT-4 and reaching the state-of-the-art among unsupervised methods, for both benchmarks.

1 Introduction

Building grounded agents that map human language instructions to a sequence of executable actions is a long-standing goal of artificial intelligence (Winograd, 1972). A promising new approach for building such agents is to use large language models to control policies in environments like web-browsers and computers (Yao et al., 2022; Murty et al., 2024; Xie et al., 2024, among others).

Unfortunately, language models struggle with such grounded instruction following out-of-the-box because LMs do not know about the myriad and ever changing interaction possibilities of different websites. For instance, on a new e-commerce website, a zero-shot LM browser agent may struggle to make a return or change order details, without expensive test-time exploration. Even simple tasks like choosing a flight can involve different UI element such as directly entering airport codes or interacting with drop-down menus, and a zero-shot agent cannot know a priori the correct thing to do.

The most common solution is to provide LM browser agents with knowledge about new web interfaces via expert demonstrations, that can either be used for in-context learning (Yao et al., 2022) or supervised fine-tuning (Lai et al., 2024; Shen et al., 2024). These demonstrations are either fully provided by human experts (Sodhi et al., 2023; Yao et al., 2022) or consist of human-generated trajectories paired with model-generated instructions (Lai et al., 2024). However, collecting human demonstrations that cover each possible use case for every website is an unattractively large, never-ending task. Thus, in this work, we propose a method for training LM browser agents in a *completely unsupervised way*, via synthetic demonstrations derived from interaction.

At a high level, our approach, NNetNav (Fig 2), uses a language model exploration policy to perform extended interactions with a website, and another language model trajectory labeler to annotate trajectories with instructions. To effectively control the exponential space of meaningful interactions, NNetNav uses the hierarchical structure of language instructions as a pruning heuristic: for exploration to discover a meaningfully complex task, trajectory prefixes must correspond to meaningful sub-tasks. Thus, during an exploration episode, if a language model cannot label trajectory prefixes (at set time-steps) with a sub-task, further exploration is automatically pruned. Imposing such a structure

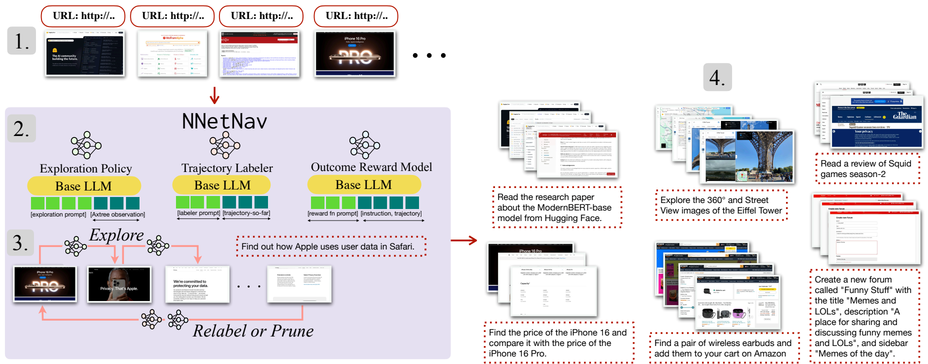

Figure 1: Given web URLs (1), `NNetNav` (2) uses a structured exploration strategy to interact with websites (3) and autonomously discover diverse (instruction, trajectory) demonstrations, as summarized in (4). To effectively prune exploration, the trajectory-so-far is periodically evaluated by a relabeling module and further exploration continues only if it can be assigned a meaningful language instruction. All components in `NNetNav` are implemented with the same zero-shot base LLM.

over search not only enhances efficiency, but also results in complex and hierarchical instructions (See Table 6 for examples). `NNetNav` prompts the same base language model for exploration, relabeling and inferring sub-tasks.

We use `Llama-3.1-70B` (Dubey et al., 2024) to collect a large scale dataset of over 10k demonstrations (around 100k state, action transitions) from 20 websites, including 15 live, in-the-wild websites, and 5 self-hosted websites from WebArena (Zhou et al., 2023). We classify these instructions into various intents and find a highly diverse range of internet use cases, including *flight booking*, *finding recipes*, *buying iPhones*, *searching for trails*, *commenting on github issues*, and *posting on Reddit* (see Fig 3 for more examples). We use these demonstrations for supervised fine-tuning of `Llama-3.1-8B`. On WebArena, our model achieves a success rate of 16.3%, outperforming zero-shot `GPT-4` by 2 points and reaching state-of-the-art performance among unsupervised methods. On WebVoyager (He et al., 2024), our best model reaches a success rate of 35.2%, outperforming zero-shot `GPT-4` by 1.7 points and all known open methods on this task to the best of our knowledge. Interestingly, we find that `NNetNav` enables effective self-training—fine-tuning a smaller LM using `NNetNav` demonstrations generated by the same model yields a 4 point absolute improvement (from 1% to 5%) on WebArena. `NNetNav` opens up interesting avenues for open-ended discovery of workflows on unknown web-interfaces, without human supervision.

2 Background

Following instructions on a web-browser is a multi-turn sequential decision making problem. Given an instruction g, a browser agent interacts with the browser by issuing a sequence of *computer control* actions $\langle a_1, a_2, \dots, a_T \rangle$ where each $a_i \in \mathcal{A}$ is drawn in response to an observation o_i. Executing an action causes a state transition based on some unknown environment dynamics, leading to a new observation o_{i+1}. The entire episode can be summarized as a *trajectory* $\tau \coloneqq \langle o_1, a_1, o_2, a_2, \dots o_{T-1}, a_T, o_T \rangle$. We formalize the instruction following agent as a mapping $\pi(a_t \mid o_t, \tau_{<t}; g)$ where $\tau_{<t} \coloneqq \langle o_1, a_1, \dots a_{t-1} \rangle$ is the trajectory so far. In our case, observations are represented as either flattened DOM trees or website accessibility trees, and $\mathcal{A}$ consists of keyboard / mouse commands that operate on elements of these trees (see Appendix A for the full action space).

LLMs for Browser Control. Recent work explores using instruction-tuned large language models (LLMs) to directly parameterize the agent. These methods typically work in settings with textual observations and action spaces. At time-step t, the agent π_{LM} is provided with the following context: the instruction g, the full action space described as a string, the current observation o_t, and some representation of the trajectory-so-far $\tau_{<t}$, typically the action history. Given this information, the LLM generates an output that is parsed into an action. Typically, the LLM output contains both a

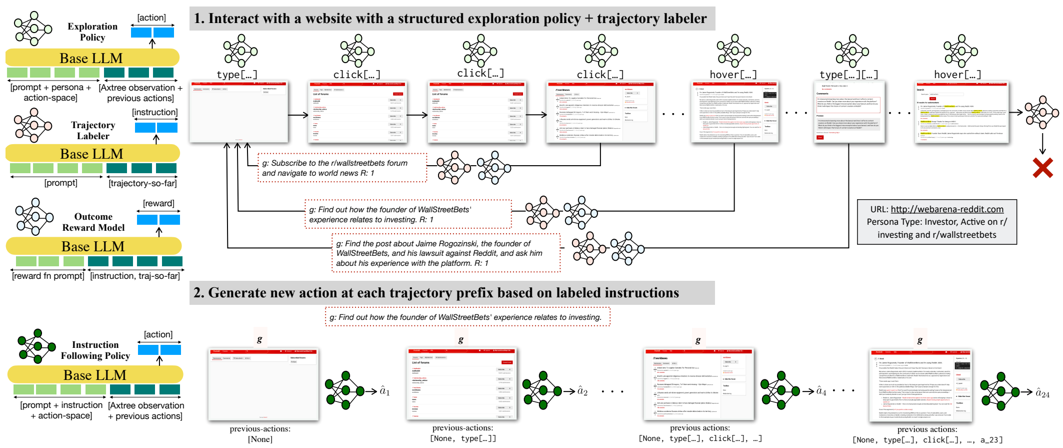

Figure 2: **Left:** `NNetNav` uses four components to interact with websites to create training examples, built out of zero-shot language models. **Right (Top):** An exploration episode on a website begins with sampling a persona, followed by generating persona-conditioned action sequences from the exploration policy. At fixed intervals, the trajectory labeler infers an instruction to describe the trajectory so far. If the resulting (instruction, trajectory) pair receives a low score from the ORM, the episode is pruned (indicated by a red cross). **Right (Bottom):** For each instruction, we retroactively generate a new action, given the (instruction, observation, previous actions) tuple to ensure that actions at each time-step correspond directly to the inferred instruction.

reasoning step r_t (e.g. *Since my task is to buy a mug, given the current state, I should click on the buy now button*), and the chosen action command a_t (e.g. *click [1234]*).

Given expert demonstrations $\{g^i, \tau^i\}$ where $\tau^i := \langle o_1^i, r_1^i, a_1^i, o_2^i, r_2^i, a_2^i \dots o_T^i \rangle$, previous work adapts LM agents using demonstrations as in-context examples (Yao et al., 2022; Shinn et al., 2023; Sun et al., 2023; Kim et al., 2023, among others) or as training data for supervised fine-tuning (Furuta et al., 2023; Lai et al., 2024; Lù et al., 2024; Patel et al., 2024). For supervised fine-tuning of π_{LM} on a dataset of demonstrations, we construct training instances $\{(g^i, \tau_{<t}^i, o_t^i), (r_t^i, a_t^i)\}$ where r_t^i, a_t^i serves as the target reasoning step and action for an intermediate context $(g^i, \tau_{<t}^i)$.

Prior Methods for Synthetic Demonstrations. Since collecting human demonstrations for browser agents is time consuming and costly, recent work uses synthetic demonstrations as training data (Lai et al., 2024; Furuta et al., 2023; Murty et al., 2024). These methods start by sampling synthetic instructions from an instruction generator (a prompted LM that takes the website landing page and an optional user persona), and use a zero-shot browser agent to convert these instructions into trajectories. Resulting demonstrations are filtered using either the ground truth reward function (Furuta et al., 2023), or using another LM outcome reward function (Lai et al., 2024; Murty et al., 2024). These methods typically fine-tune smaller LMs using synthetic demonstrations from larger LMs.

Such *instruction-first* methods for data collection face several challenges. First, synthetic instructions in these demonstrations are sampled from an ungrounded LM prior that generates only plausible[1] instructions without ensuring feasibility; *e.g.*, an instruction such as *Delete the first post on r/callofdutyfans* for reddit is plausible, but not always feasible. Second, generated instructions are limited to those that reference visible features of the website; *e.g.*, given the landing page of a github-like platform, no LM prior can generate instructions like *Find information about Eric Bailey's contributions to the byteblaze project*, which require knowing about deeply embedded website-specific entities like *Eric Bailey*. Finally, these methods provide no control over the complexity of instructions, and rely entirely on the LM or bespoke prompts to generate complex instructions.

[1]We use the term *plausible* for instructions that match a website's genre or intended use. For example, searching for clothes on a retail site or checking notifications on a social media platform. Not all plausible instructions are feasible.

3 OUR APPROACH

Instead of starting with a sampled instruction, we start by sampling an *interaction* first, and then retroactively labeling it into an instruction that is feasible by definition. `NNetNav` (Fig 2) is an *interaction-first* method for constructing demonstrations: An exploration policy interacts with a browser in a structured manner to sample long trajectories which are retroactively labeled into instructions (§3.2). We then post-process each trajectory to add post-hoc actions corresponding to the generated instructions.

3.1 LM COMPONENTS

All components in `NNetNav` are implemented with a zero-shot instruction-tuned LLM, with different prompts (see Appendix A for prompts).

Exploration Policy. To interact with the environment, we use an exploration policy π_{explore}, implemented as -a prompted language model, similar to π_{LM}. Additionally, to simulate a diverse set of behaviors from users and improve the diversity of resulting trajectories, we seed each episode with a string description of a plausible user persona for the given website (Shanahan et al., 2023; Argyle et al., 2023). At each time-step, π_{explore} is provided with the following context: a user persona, the list of available actions, the current observation o_t, and the action history. The output of π_{explore} is then parsed into an action.

Summarizing Trajectory changes. Actions issued by π_{explore} result in a new observation in the environment. We summarize this change as a short string description via another module Δ_{LM}, implemented using a language model. In particular, for any state o_t, action a_t and the resulting next state o_{t+1}, $\delta_t = \Delta_{\text{LM}}(o_t, a_t, o_{t+1})$ produces a string-valued description of the changes in the observation as a result of the action. For a trajectory τ, we denote the sequence of state changes as δ_τ.

Trajectory Labeler. Given δ_τ, the trajectory labeler Lf_{LM} produces a plausible instruction $\hat{g} = \text{Lf}_{\text{LM}}(\delta_\tau)$ that the agent could have followed to produce the given interaction.

Outcome Reward Model. Given $\hat{g}$ and δ_τ, the outcome reward model (ORM) assigns a reward $s_{\text{LM}}(\hat{g}, \delta_\tau) \in \{0, 1\}$, based on the degree to which state changes correspond to the given instruction $\hat{g}$.

3.2 SAMPLING DEMONSTRATIONS VIA INTERACTIONS

At specific time steps $t \in \{t_1, t_2, \ldots, t_{\text{max}}\}$, we apply a pruning heuristic to retroactively label the current trajectory. Given a partial trajectory $\tau_{<t}$ after interacting with the environment for t steps, we compute a sub-task annotation $\hat{g} = \text{Lf}_{\text{LM}}(\delta_{\tau_{<t}})$. If this sub-task receives no reward, i.e., $s_{\text{LM}}(\hat{g}, \delta_{\tau_{<t}}) = 0$, we prune the episode and sample a new rollout. Otherwise, we store $(\hat{g}, \tau_{<t})$ as a synthetic demonstration and continue exploration. Each episode typically generates multiple such demonstrations.

Post-processing with an Agent Policy. Actions at each time-step in our our demonstration set are a result of un-directed exploration, and therefore might not be optimal for the retroactively generated instruction. Thus, we post-hoc annotate each state with a new action that directly corresponds to the generated instruction. Concretely, given every $(\hat{g}, o_i, \tau_{<t})$ tuple in our synthetic demonstration set, we use π_{LM} to output a suitable action $\hat{a}_i$ given the instruction $\hat{g}$ and current observation o_i.

BofK sampling (Optional). To further boost the quality of trajectories, we optionally use best-of-K (BofK) sampling. In particular, given `NNetNav` generated instructions, we sample $K - 1$ additional trajectories, with π_{LM} using the same base LLM. Then, for each instruction, we use our ORM to score each of the $K - 1$ trajectories and the original trajectory, and pair the best trajectory with the given instruction, breaking ties arbitrarily.

4 Main Experiments

4.1 Collecting Demonstrations in the Wild

We apply NNetNav on 20 websites to collect a dataset of over 10,000 demonstrations. We consider 15 live websites (same set as He et al. 2024): Allrecipes, Amazon, Apple, ArXiv, BBC News, Booking, Cambridge Dictionary, Coursera, ESPN, GitHub, Google Flights, Google Map, Google Search, Huggingface, and Wolfram Alpha, and 5 self-hosted websites from WebArena (WA; Zhou et al., 2023).

We use instruct-tuned `Llama-3.1-70b` as the base LLM for all components in NNetNav, with t_{max} set to 40, running NNetNav pruning every 4 time-steps at {4, 8, 12, 16, ..., 40}. Additionally, we perform BofK sampling with $K = 3$, using π_{LM} (with the same `Llama-3.1-70b` base model). While we only consider text based browser agents in this work, we release both accessibility tree strings as well as browser screenshots at each time step, to support future work on multi-modal browser agents.

Difficulty	**NNetNav (WA)**	**NNetNav (Live)**
Easy	498	1448
Medium	2532	2369
Hard	1164	1204
Very-Hard	501	556
Total	4695	5577

Table 1: We report the breakdown of NNetNav demonstrations into categories defined based on the number of actions in the trajectory.

Diversity and Complexity. To evaluate diversity in resulting instructions, we cluster them by intent for each website. We obtain these intents through a two-step procedure—we input instructions for each website into `GPT-4o`, prompting it to identify common intents, and then classify each instruction into one of these intents in a second forward pass. On average, we identify 21 intents per website for self-hosted websites and 25 for live websites. Analyzing the distribution of these intents, we observe an average perplexity (PPL) of 13.5 for self-hosted sites and 16.2 for live websites. Higher perplexity suggests a more evenly distributed set of intents, indicating substantial diversity in the collected demonstrations. We provide a visual representation of this distribution as a sunburst plot in Appendix D.

To analyze the complexity of demonstrations, we categorize each demonstration into one of four levels based on the number of action sequences: *easy* (fewer than 5 actions), *medium* (5 to 10 actions), *hard* (10 to 20 actions), and *very hard* (over 20 actions). Table 1 presents the distribution of demonstrations across these categories, showing a substantial number of complex demonstrations.

4.2 Finetuning: Details and Results

We perform supervised fine-tuning of the smaller instruct-tuned `Llama-3.1-8B` with NNetNav demonstrations. To measure transfer between knowledge learned from live websites and self-hosted WebArena websites, we fine-tune on: only WebArena websites (Llama8B-NNetNav-WA), only live websites (Llama8B-NNetNav-Live), and all websites together (Llama8B-NNetNav-All).

As described in Section 2, each demonstration expands into multiple training instances, resulting in a total of 100k training examples for the full dataset. We fine-tune for 2 epochs with a batch size of 128, truncating the max sequence length to 20000, with a learning rate of 2e-5, that is warmed with a linear scheduler over 500 gradient updates (more details can be found in Appendix C). We use open-instruct (Wang et al., 2023) for fine-tuning, and set up local inference servers using VLLM (Kwon et al., 2023). During inference, we sample with a temperature of 0.01 and perform nucleus sampling (Holtzman et al., 2019) with top-p set to 0.9.

Agent	#Params	WebArena SR	WebVoyager SR	Human Supervision Used?
Using Closed Models				
GPT-4 (Zhou et al., 2023; He et al., 2024)	Unknown	14.1	33.5	✗
GPT-4-AWM (Wang et al., 2024)	Unknown	35.5	-	✗
GPT-4 + LLama-70b (Shen et al., 2024)	Unknown	50.0	-	✓
Using Open Models				
`Llama-3.1-8b`	8B	1.0	4.4	✗
Lai et al. (2024)	7B	2.5	-	✗
Ou et al. (2024)	7B	6.3	-	✗
Patel et al. (2024)	72B	9.4	-	✗
LLaVa-7B PAE + Claude (Zhou et al., 2024)	7B	-	22.3	✗
LLaVa-34B PAE + Claude (Zhou et al., 2024)	34B	-	33.0	✗
Qwen2.5-7B-AgentTrek (Xu et al., 2024)	7B	10.5	-	✗
Qwen2.5-32B-AgentTrek (Xu et al., 2024)	32B	16.3	-	✗
`Llama8B-NNetNav-WA` **(Ours)**	8B	**16.3**	28.1	✗
`Llama8B-NNetNav-Live` **(Ours)**	8B	9.5	**35.2**	✗
`Llama8B-NNetNav-All` **(Ours)**	8B	14.9	34.1	✗

Table 2: We present average success rate (SR) on browser tasks from WebArena and WebVoyager for various approaches, along with key details such as model size, the use of open LLMs and human supervision. For Lai et al. (2024), we report results from the setting that does not use human supervision. Zero-shot `GPT-4` results are sourced from Zhou et al. (2023) and He et al. (2024). The last three rows report the performance of our fine-tuned `Llama-3.1-8b` agents, which achieve state-of-the-art results, outperforming zero-shot `GPT-4` and outperforming or matching prior open-model approaches with significantly fewer parameters, across both benchmarks.

Benchmarks. We evaluate models on 812 tasks from WebArena (Zhou et al., 2023) and 557 tasks from WebVoyager (He et al., 2024), omitting tasks in Google Flights and Booking, as they are no longer feasible (following Zhou et al., 2024). For WebArena, we report averaged success rate (SR) across all tasks based on the provided evaluator that measures functional correctness. For WebVoyager, we use the author-provided script that uses GPT-4V to judge success based on instructions and browser screenshots at each time step. We report the average across all websites.

Results. We report our results in Table 2, where we present prior results from using closed models (typically `GPT-4o`) as well as with open models. On WebArena, both `Llama8B-NNetNav-WA` and `Llama8B-NNetNav-All` outperform zero-shot `GPT-4o`, with our best model achieving state-of-the-art performance among unsupervised methods. On WebVoyager, `Llama8B-NNetNav-Live` and `Llama8B-NNetNav-All` surpass zero-shot `GPT-4o`, establishing a new state-of-the-art among open-source methods. Notably, they outperform the previous best OSS result from Zhou et al. (2024), which relied on a significantly larger 34B-parameter vision-language model (VLM) and a closed-model verifier. Interestingly, we find that `Llama8B-NNetNav-WA`, which is trained exclusively on WebArena websites, exhibits poor transfer to live websites. We analyze cross-website transfer next.

4.3 Cross-Website Transfer

We present per-website success rates of our fine-tuned models across all 18 websites in Table 3. For WebArena websites, by comparing columns 2 and 3, we find that 3 out of 5 websites benefit from incorporating in-domain data. By comparing columns 1 and 3, we observe an average performance drop of 1.8 points, with the most significant decrease on the *Maps* domain. This decline is likely due to the semantic search capabilities in *Google Maps*, which are absent in WebArena *Maps*, necessitating more complex query formulation. For live websites, fine-tuning on in-domain live website data improves performance on 10 out of 13 domains, as indicated by comparing columns 1 and 3. The effect of incorporating out-of-domain WebArena data, however, is mixed. While it results in negative transfer for 7 websites and positive transfer for 6, the overall average performance decreases by 1.3 points. Notable gains are observed in *ESPN*, *Apple*, and *GitHub*, suggesting potential synergies when fine-tuning on closely related domains.

Overall, fine-tuning with in-domain website data improves performance on 13 out of 18 websites. These findings underscore the importance of learning from unsupervised interaction on real websites,

Website	NNetNav (WA)	NNetNav (Live)	NNetNav (Live+WA)
Self-hosted Websites (WebArena)			
Reddit	**26.3**	9.6	25.4
Gitlab	**18.4**	5.6	16.8
Maps	**15.6**	14.8	10.9
CMS	**11.5**	5.5	9.9
Shopping	**13.0**	9.9	**13.0**
Live Websites (WebVoyager)			
Allrecipes	26.7	**37.8**	29.5
Amazon	24.4	**43.9**	34.1
Apple	32.6	27.9	**34.9**
ArXiV	27.9	**46.5**	44.2
BBC News	33.3	**42.9**	28.6
Cambridge Dictionary	46.5	**58.1**	48.8
Coursera	**47.6**	45.2	42.9
ESPN	20.5	22.7	**27.3**
GitHub	12.2	17.1	**19.5**
Google Maps	34.1	**46.3**	43.9
Google Search	0.0	2.7	**6.2**
Huggingface	**30.2**	18.6	**30.2**
Wolfram Alpha	26.1	43.5	**45.7**

Table 3: Per-website success rates on all websites, using a `Llama-3.1-8b` agent fine-tuned on (1) the WebArena subset of `NNetNav`, (2) the live website subset of `NNetNav`, and (3) all demonstrations. On WebArena, incorporating in-domain data improves performance on 3 out of 5 websites (comparing columns 2 and 3). For live websites, incorporating in-domain data improves performance for 10 out of 13 websites (comparing columns 1 and 3). These results highlight the importance of scalable methods to enable training on diverse websites.

as relying solely on human-labeled trajectories from a limited set of simulated websites may be insufficient for developing generalist web agents.

5 Controlled Experiments

We conduct controlled experiments on a smaller scale to compare `NNetNav` with baselines. In addition to evaluating on WebArena, we also consider MiniWoB++ (Shi et al., 2017; Liu et al., 2018). MiniWoB++ is a dataset of synthetic web-interfaces with a shared action space. Tasks on MiniWoB++ range from clicking on buttons to complex tasks like making a booking on a website. We use a subset of 8 complex tasks from MiniWoB++ as a toy benchmark to evaluate our method. We use the `bid`-based action space from BrowserGym (Drouin et al., 2024), consisting of 12 actions, and a DOM based observation space. Due to its synthetic nature, MiniWoB++ comes with an automatic reward function. We report the mean reward over 20 random seeds for each task, similar to Drouin et al. (2024).

5.1 Experimental Settings

As before, we evaluate a `Llama-3.1-8b` based browser agent under the following settings:

1. **Zero-Shot:** A baseline zero-shot agent, prompted using chain-of-thought prompting (Wei et al., 2022). Next, we consider various fine-tuned models.
2. **SFT (Instruction-First):** Supervised fine-tuning of the `Llama-3.1-8b` agent using data collected via instruction-first sampling. Here, we use the same reward model for filtering demonstrations as `NNetNav`, and also sample the same number of demonstrations for fair comparison.
3. **SFT (`NNetNav`):** Supervised fine-tuning of the `Llama-3.1-8b` agent with demonstrations collected via `NNetNav`.
4. **SFT (`NNetNav` + Distil.):** Ablation, where we only retain instructions found via `NNetNav` and re-generate trajectories by prompting the same large LM as an agent. We use this setting to isolate performance improvements attributable to `NNetNav` trajectories.

For these small scale experiments, we use `gpt-4o-mini-2024-07-18` as the base LLM for both `NNetNav` and instruction-first methods. For Instruction-first data collection, we sample 50 instructions per website for WebArena, and 80 instructions per interface in MiniWoB++, and prompt the instruction generator with the landing page as well as a persona (to improve diversity). For `NNetNav`, we use our exploration policy to generate 50 episodes per website for WebArena, and 80 episodes per interface for MiniWoB++. We set $T_{\max}$ to 40 for WebArena, and 20 for MiniWoB++. For both MiniWoB++ and WebArena, we apply the pruning function every 4 time-steps. We use 16 persona types per website for WebArena, and 10 persona types per web-interface for MiniWoB++.

Model Setting	**MiniWoB++**	**WebArena**
Zero-Shot	0.28	1.0
SFT (Instruction-First)	0.28	4.2
SFT (`NNetNav`)	**0.48**	**7.2**
SFT (`NNetNav` + Distil.)	0.36	6.0

Table 4: Controlled evaluation of `NNetNav` with instruction-first methods. We present results for MiniWoB++ and WebArena, averaged across domain, reporting mean reward for MiniWoB++ and task success rate (SR) for WebArena. Fine-tuning with `NNetNav` leads to the largest improvements: from 28% to 48% on MiniWoB++; from 1% to 7.2% on WebArena.

5.2 Results

`NNetNav` outperforms instruction-first methods. We report results from all settings in Table 4. Fine-tuning `Llama-3.1-8b` using synthetic demonstrations generated by `NNetNav` yields significant improvements: an increase of 20 points on MiniWoB++ and over 6 points on WebArena. Notably, `NNetNav` outperforms instruction-first methods by a substantial margin, with gains of 12 points on MiniWoB++ and 1.2 points on WebArena. Interestingly, SFT (`NNetNav`) outperforms SFT (`NNetNav` + Distil.) on both MiniWoB++ and WebArena. This difference likely stems from the distinct procedures used to generate trajectories. In `NNetNav`, the model first acts, and the corresponding instruction is inferred afterward through a hindsight procedure. In contrast, `NNetNav` + Distil. provides the instruction upfront, sampling the trajectory later.

Self-training with `NNetNav`. Can `NNetNav` demonstrations from an LM be used for improving the *same* LM agent? To answer this, we collect another set of `NNetNav` demonstrations on WebArena, using `Llama-3.1-8b` as the base LM for data collection. Given the limitations of this smaller model, we anticipate fewer meaningful interactions. To compensate, we increase the number of episodes to 200 episodes per website, resulting in 302 demonstrations which we use for fine-tuning the same `Llama-3.1-8b` agent. From results in Table 5, we find improvements of 4.3 points on WebArena.

Domain	**Zero-Shot**	**Self-Train (`NNetNav`)**
Shopping	3.8	**15.4**
CMS	0.0	0.0
Reddit	0.0	0.0
Gitlab	0.0	0.0
Maps	0.0	**7.1**
Avg.	1.0	**5.3**

Table 5: We generate `NNetNav` demonstrations using `Llama-3.1-8b`, which we use for supervised fine-tuning of an agent based on the same LM, and find significant improvements on WebArena from 1% to 5.3%.

6 Related Work

Language Conditioned Digital Assistants. Mapping instructions to actions in digital environments has been a long-standing goal in natural language understanding (Allen et al., 2007; Branavan et al., 2009). Most pre-LLM approaches for this rely on expert demonstrations for behavioral cloning (Chen & Mooney, 2011; Humphreys et al., 2022), along with appropriately shaped reward functions (Branavan et al., 2009; Liu et al., 2018; Misra et al., 2017, among others). Here, learning is driven purely by synthetic demonstrations derived via (language model) exploration of websites.

Linguistic Priors for Exploration. Several prior works have used natural language priors to inform exploration for sequential decision making. Harrison et al. (2017) use a *trained model* of associations between language and state/action pairs to guide exploration during policy learning. Mu et al. (2022) use language annotations of states to train a goal generator module that provides intrinsic rewards for achieving generated goals. Similarly, Du et al. (2023) constrain exploration towards goals generated by a pre-trained LLM at each intermediate state of an agent. In constrast, `NNetNav` biases exploration through two news ways of using language priors. First, we use natural language as a way to filter meaningful interactions. Second, we use it as a pruning heuristic to navigate the potentially exponential search space of these interactions.

Training Data for LLM browser agents. LLMs have shown strong performance over a wide range of language understanding tasks, and are increasingly being used to interpret language in grounded contexts such as browsers (Yao et al., 2022; Lai et al., 2024; Wang et al., 2024; Patel et al., 2024; Lù et al., 2024, among others). Many of these approaches rely on human demonstrations, either for in-context learning (Yao et al., 2022; Sodhi et al., 2023; Kim et al., 2023) or for finetuning (Lù et al., 2024; Shen et al., 2024). Since human demonstrations are costly, recent work trains LLM agents through synthetic demonstrations generated using instruction-first methods (Lai et al., 2024; Patel et al., 2024). One exception is Murty et al. (2024), which introduces an interaction-first method for generating synthetic demonstrations for in-context learning. Despite its novelty, their approach does not scale well to real websites due to the lack of a mechanism for effective exploration in environments with many possible interactions. In contrast, `NNetNav` also follows an interaction-first approach but improves efficiency by leveraging linguistically motivated pruning to navigate the space of meaningful interactions.

7 Conclusion

We propose `NNetNav`, a method for unsupervised interaction with websites "in-the-wild" that enables training browser agents with synthetic demonstrations. `NNetNav` flips the standard paradigm of synthetic data generation by first interacting with a website to generate trajectories and then hindsight relabeling trajectories into instructions. Real websites have a prohibitively large set of possible interactions; `NNetNav` searches over this space efficiently using a pruning function inspired by the hierarchical structure of language instructions: any complex instruction consists of language describable sub-tasks and so, if during an interaction a relabeling module cannot infer a meaningful sub-task for the trajectory-so-far, further exploration is pruned. We apply `NNetNav` to collect a diverse and complex set of 10k demonstrations from 15 live-websites and 5 self-hosted websites. We use these demonstrations for supervised finetuning of a small, `Llama-3.1-8b` model, achieving state-of-the-art results for unsupervised methods on both the WebArena and WebVoyager, surpassing zero-shot `GPT-4` by 1.7 to 2.2 points. `NNetNav` opens up the possibility of scaling up training data for generalist web agents across a broad range of web interfaces without any human intervention.

Impact Statement

The deployment of unsupervised exploration with LLM agents on live websites has real-world implications, including website overload, unintended interactions, and the propagation of biases. To mitigate potential disruptions to websites, we constrain our agents to a maximum of 10 parallel instances, enforce a 0.5-second delay between actions, and prohibit login or content submission on live websites. We suggest that anyone using our work closely monitor these agents and establish robust monitoring frameworks to detect unintended behaviors and ensure compliance with ethical guidelines. Additionally, training agents on NNetNav data from live websites can reinforce biases present in web content. We urge practitioners to conduct thorough bias audits before deployment.

REFERENCES

James Allen, Nathanael Chambers, George Ferguson, Lucian Galescu, Hyuckchul Jung, Mary Swift, and William Taysom. Plow: a collaborative task learning agent. In *Proceedings of the 22nd National Conference on Artificial Intelligence - Volume 2*, AAAI'07, pp. 1514–1519. AAAI Press, 2007. ISBN 9781577353232.

Lisa P Argyle, Ethan C Busby, Nancy Fulda, Joshua R Gubler, Christopher Rytting, and David Wingate. Out of one, many: Using language models to simulate human samples. *Political Analysis*, 31(3):337–351, 2023.

S.R.K. Branavan, Harr Chen, Luke Zettlemoyer, and Regina Barzilay. Reinforcement learning for mapping instructions to actions. In Keh-Yih Su, Jian Su, Janyce Wiebe, and Haizhou Li (eds.), *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pp. 82–90, Suntec, Singapore, August 2009. Association for Computational Linguistics. URL https://aclanthology.org/P09-1010.

David Chen and Raymond Mooney. Learning to interpret natural language navigation instructions from observations. *Proceedings of the AAAI Conference on Artificial Intelligence*, 25(1):859–865, Aug. 2011. doi: 10.1609/aaai.v25i1.7974. URL https://ojs.aaai.org/index.php/AAAI/article/view/7974.

Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H. Laradji, Manuel Del Verme, Tom Marty, David Vazquez, Nicolas Chapados, and Alexandre Lacoste. WorkArena: How capable are web agents at solving common knowledge work tasks? In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 11642–11662. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/drouin24a.html.

Yuqing Du, Olivia Watkins, Zihan Wang, Cédric Colas, Trevor Darrell, Pieter Abbeel, Abhishek Gupta, and Jacob Andreas. Guiding pretraining in reinforcement learning with large language models. *arXiv preprint arXiv:2302.06692*, 2023.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The Llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Hiroki Furuta, Ofir Nachum, Kuang-Huei Lee, Yutaka Matsuo, Shixiang Shane Gu, and Izzeddin Gur. Multimodal web navigation with instruction-finetuned foundation models. *arXiv preprint arXiv:2305.11854*, 2023.

Brent Harrison, Upol Ehsan, and Mark O Riedl. Guiding reinforcement learning exploration using natural language. *arXiv preprint arXiv:1707.08616*, 2017.

Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. Webvoyager: Building an end-to-end web agent with large multimodal models. *arXiv preprint arXiv:2401.13919*, 2024.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*, 2019.

Peter C Humphreys, David Raposo, Tobias Pohlen, Gregory Thornton, Rachita Chhaparia, Alistair Muldal, Josh Abramson, Petko Georgiev, Adam Santoro, and Timothy Lillicrap. A data-driven approach for learning to control computers. In *International Conference on Machine Learning*, pp. 9466–9482. PMLR, 2022.

Geunwoo Kim, Pierre Baldi, and Stephen McAleer. Language models can solve computer tasks. *arXiv preprint arXiv:2303.17491*, 2023.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Hanyu Lai, Xiao Liu, Iat Long Iong, Shuntian Yao, Yuxuan Chen, Pengbo Shen, Hao Yu, Hanchen Zhang, Xiaohan Zhang, Yuxiao Dong, and Jie Tang. Autowebglm: Bootstrap and reinforce a large language model-based web navigating agent, 2024.

Evan Zheran Liu, Kelvin Guu, Panupong Pasupat, Tianlin Shi, and Percy Liang. Reinforcement learning on web interfaces using workflow-guided exploration. In *International Conference on Learning Representations*, 2018.

Xing Han Lù, Zdeněk Kasner, and Siva Reddy. Weblinx: Real-world website navigation with multi-turn dialogue. *arXiv preprint arXiv:2402.05930*, 2024.

Dipendra Misra, John Langford, and Yoav Artzi. Mapping instructions and visual observations to actions with reinforcement learning. *arXiv preprint arXiv:1704.08795*, 2017.

Jesse Mu, Victor Zhong, Roberta Raileanu, Minqi Jiang, Noah Goodman, Tim Rocktäschel, and Edward Grefenstette. Improving intrinsic exploration with language abstractions. *Advances in Neural Information Processing Systems*, 35:33947–33960, 2022.

Shikhar Murty, Christopher D Manning, Peter Shaw, Mandar Joshi, and Kenton Lee. BAGEL: Bootstrapping agents by guiding exploration with language. In Ruslan Salakhutdinov, Zico Kolter, Katherine Heller, Adrian Weller, Nuria Oliver, Jonathan Scarlett, and Felix Berkenkamp (eds.), *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 36894–36910. PMLR, 21–27 Jul 2024. URL https://proceedings.mlr.press/v235/murty24a.html.

Tianyue Ou, Frank F Xu, Aman Madaan, Jiarui Liu, Robert Lo, Abishek Sridhar, Sudipta Sengupta, Dan Roth, Graham Neubig, and Shuyan Zhou. Synatra: Turning indirect knowledge into direct demonstrations for digital agents at scale. *arXiv preprint arXiv:2409.15637*, 2024.

Ajay Patel, Markus Hofmarcher, Claudiu Leoveanu-Condrei, Marius-Constantin Dinu, Chris Callison-Burch, and Sepp Hochreiter. Large language models can self-improve at web agent tasks. *arXiv preprint arXiv:2405.20309*, 2024.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16. IEEE, 2020.

Murray Shanahan, Kyle McDonell, and Laria Reynolds. Role play with large language models. *Nature*, 623(7987):493–498, 2023.

Junhong Shen, Atishay Jain, Zedian Xiao, Ishan Amlekar, Mouad Hadji, Aaron Podolny, and Ameet Talwalkar. Scribeagent: Towards specialized web agents using production-scale workflow data. *arXiv preprint arXiv:2411.15004*, 2024.

Tianlin Shi, Andrej Karpathy, Linxi Fan, Jonathan Hernandez, and Percy Liang. World of bits: An open-domain platform for web-based agents. In *International Conference on Machine Learning*, pp. 3135–3144. PMLR, 2017.

Noah Shinn, Beck Labash, and Ashwin Gopinath. Reflexion: an autonomous agent with dynamic memory and self-reflection. *arXiv preprint arXiv:2303.11366*, 2023.

Paloma Sodhi, SRK Branavan, and Ryan McDonald. Heap: Hierarchical policies for web actions using llms. *arXiv preprint arXiv:2310.03720*, 2023.

Haotian Sun, Yuchen Zhuang, Lingkai Kong, Bo Dai, and Chao Zhang. Adaplanner: Adaptive planning from feedback with language models. *arXiv preprint arXiv:2305.16653*, 2023.

Yizhong Wang, Hamish Ivison, Pradeep Dasigi, Jack Hessel, Tushar Khot, Khyathi Raghavi Chandu, David Wadden, Kelsey MacMillan, Noah A. Smith, Iz Beltagy, and Hannaneh Hajishirzi. How far can camels go? exploring the state of instruction tuning on open resources, 2023.

Zora Zhiruo Wang, Jiayuan Mao, Daniel Fried, and Graham Neubig. Agent workflow memory. *arXiv preprint arXiv:2409.07429*, 2024.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837. Curran Associates, Inc., 2022.

Terry Winograd. Understanding natural language. *Cognitive Psychology*, 3(1):1–191, 1972. ISSN 0010-0285. doi: https://doi.org/10.1016/0010-0285(72)90002-3. URL https://www.sciencedirect.com/science/article/pii/0010028572900023.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, et al. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. *arXiv preprint arXiv:2404.07972*, 2024.

Yiheng Xu, Dunjie Lu, Zhennan Shen, Junli Wang, Zekun Wang, Yuchen Mao, Caiming Xiong, and Tao Yu. Agenttrek: Agent trajectory synthesis via guiding replay with web tutorials. *arXiv preprint arXiv:2412.09605*, 2024. URL https://arxiv.org/abs/2412.09605.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2022.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, et al. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.

Yifei Zhou, Qianlan Yang, Kaixiang Lin, Min Bai, Xiong Zhou, Yu-Xiong Wang, Sergey Levine, and Erran Li. Proposer-agent-evaluator (pae): Autonomous skill discovery for foundation model internet agents. *arXiv preprint arXiv:2412.13194*, 2024.

A PROMPTS FOR LM COMPONENTS

A.1 MINIWOB++

We start by presenting all prompts for MiniWoB++. The action space for MiniWob++ is:

Listing 1: Action Space

```
noop(wait_ms: float = 1000)
    Examples:
        noop()
        noop(500)

scroll(delta_x: float, delta_y: float)
    Examples:
        scroll(0, 200)
        scroll(-50.2, -100.5)

fill(bid: str, value: str)
    Examples:
        fill('237', 'example value')
        fill('45', 'multi-line\nexample')
        fill('a12', 'example with "quotes"')

select_option(bid: str, options: str | list[str])
    Examples:
        select_option('a48', 'blue')
        select_option('c48', ['red', 'green', 'blue'])

click(bid: str, button: Literal['left', 'middle', 'right'] = 'left', modifiers: list[typing
.Literal['Alt', 'Control', 'Meta', 'Shift']] = [])
    Examples:
        click('a51')
        click('b22', button='right')
        click('48', button='middle', modifiers=['Shift'])

dblclick(bid: str, button: Literal['left', 'middle', 'right'] = 'left', modifiers: list[
typing.Literal['Alt', 'Control', 'Meta', 'Shift']] = [])
    Examples:
        dblclick('12')
        dblclick('ca42', button='right')
        dblclick('178', button='middle', modifiers=['Shift'])

hover(bid: str)
    Examples:
        hover('b8')

press(bid: str, key_comb: str)
    Examples:
        press('88', 'Backspace')
        press('a26', 'Control+a')
        press('a61', 'Meta+Shift+t')

focus(bid: str)
    Examples:
        focus('b455')

clear(bid: str)
    Examples:
        clear('996')

drag_and_drop(from_bid: str, to_bid: str)
    Examples:
        drag_and_drop('56', '498')

upload_file(bid: str, file: str | list[str])
    Examples:
        upload_file('572', 'my_receipt.pdf')
        upload_file('63', ['/home/bob/Documents/image.jpg', '/home/bob/Documents/file.zip
        '])

Only a single action can be provided at once. Example:
fill('a12', 'example with "quotes"')

If you are done exploring, you can issue the stop action: ```stop```
```

```
Here is an example with chain of thought of a valid action when clicking on a button: "In
order to accomplish my goal I need to click on the button with bid 12. In summary, the next
 action I will perform is ```click("12")```
```

This is then directly used for various prompts as `{action_str}`.

Listing 2: Prompt for the Exploration Policy π_{explore}

```
You are an autonomous intelligent agent tasked with performing tasks on a web interface.
Your objective is to simulate a task that a person might request, by interacting with the
interface through the use of specific actions.

Here's the information you'll have:
DOM Representation: This is the current webpage's Document Object Model (DOM)
representation as a string.
The previous action: This is the action you just performed. It may be helpful to track your
 progress.
Trajectory: This is a sequence of natural language descriptions of the agent's interaction
with the web-browser.
Person Description: The description of a specific kind of person whose task you are
supposed to simulate.

You can perform the following actions: {action_str}

To be successful, it is very important to follow the following rules:
1. You should only issue an action that is valid given the current observation.
2. You should only issue one action at a time.
3. You should reason step by step and then issue the next action.
4. Make sure to wrap your action in a code block using triple backticks.
5. The DOM / Accessibility Tree only shows the visible part of the webpage. If you need to
interact with elements that are not visible, you can scroll to them using the scroll action
. Often submit buttons are not visible and are at the bottom of the page. To scroll to the
bottom of the page, use the scroll action with a large value for the y coordinate.
6. To generate an interesting task, make sure you issue atleast 4 actions before stopping.
More interesting tasks typically involve more interactions with the browser.
7. You can issue atmost 20 actions before stopping, but feel free to output the stop action
 early if you want to stop exploring. Don't generate anything after stop.
```

Listing 3: Prompt for Δ_{LM}

```
You are given the output of an action taken by an autonomous intelligent agent navigating a
 web-interface to fulfill a task given by a user.  Your objective is to produce a
description of the changes made to the state of the browser.

Here's the information you'll have:
Initial state of the browser as a DOM representation: This is the webpage's Document Object
 Model (DOM) representation as a string.
Final state of the browser as a DOM representation: This is the DOM representation after
the agent took the action.

The action taken by the agent: This is the action taken by the agent to change the state of
 the browser.

The actions the agent can take come from the following categories: {action_str}

To be successful, it is very important to follow the following rules:
1. Explictly think about the various features on the website and how the interaction with
the website changed these features
2. Provide the description of changes in one or two sentences.
3. Strictly follow the format "State change: <your-answer>" for your output
```

Listing 4: Prompt for the Trajectory Labeling function Lf_{LM}

```
Given a task from a user, an autonomous intelligent agent carries out a sequence of actions
 on a web-interface.

The actions the agent can take fall under the following categories: {action_str}

Your objective is to guess the instruction the user gave, given the following information:
```

```
Trajectory: This is a sequence of natural language descriptions of the agent's interaction
with the web-browser.

To be successful, it is very important to follow the following rules:
1. Explictly think about how the trajectory is a valid way to achieve the instruction,
before outputing the instruction.
2. Start by thinking by outputing Thought: <your-reasoning>.
3. End your answer by strictly following the format "Instruction: <your-answer>" for your
output.
```

Listing 5: Prompt for the reward function s_{LM}

```
An autonomous intelligent agent navigating a web browser is given an instruction by a user.
 Your objective is to give a score to the agent based on how well it completed its task.
Your score must be on the scale of 1 to 5. Give a score of 5 only when there are no errors.
 To do this task you are provided with the following information:

Instruction: This is the natural language instruction given to the agent.
Trajectory: This is a sequence of natural language descriptions of the agent's interaction
with the web-browser.

To be successful, it is very important to follow the following rules:
1. Explictly think about what is needed to follow the instruction correctly on the website
and if the trajectory reflects these steps.
2 Give a score of 4 if there are very minor errors, or if the task was more than 70%
completed. Give a score of 3 (or below) if the model made very little progress towards the
given instruction or if there are major errors.
3. Start by thinking by outputing Thought: <your-reasoning>.
4. End your answer by strictly following the format "Reward: <your-answer>" for your output
```

Listing 6: Prompt for the base LLM agent π_{LM}

```
You are an autonomous intelligent agent tasked with performing tasks on a web interface.
These tasks will be accomplished through the use of specific actions you can issue.

Here's the information you'll have:
DOM Representation: This is the current webpage's Document Object Model (DOM)
representation as a string.
The user's objective: This is the task you're trying to complete.
The previous action: This is the action you just performed. It may be helpful to track your
 progress.

You can perform the following actions: {action_str}

To be successful, it is very important to follow the following rules:
1. You should only issue an action that is valid given the current observation
2. You should only issue one action at a time.
3. You should follow the examples to reason step by step and then issue the next action.
4. Make sure to wrap your action in a code block using triple backticks.
5. The DOM / Accessibility Tree only shows the visible part of the webpage. If you need to
interact with elements that are not visible, you can scroll to them using the scroll action
. Often submit buttons are not visible and are at the bottom of the page. To scroll to the
bottom of the page, use the scroll action with a large value for the y coordinate.
6. Issue stop action when you think you have achieved the objective. Don't generate
anything after stop.
```

A.2 Prompts for WebArena and Live Websites

Next, we present all prompts for running policies on self-hosted WebArena websites and live websites. The action space is:

Listing 7: Action Space

```
Page Operation Actions:
`click [id]`: This action clicks on an element with a specific id on the webpage.
`type [id] [content] [press_enter_after=0|1]`: Use this to type the content into the field
with id. By default, the "Enter" key is pressed after typing unless press_enter_after is
set to 0.
`hover [id]`: Hover over an element with id.
```

```
`press [key_comb]`:  Simulates the pressing of a key combination on the keyboard (e.g.,
Ctrl+v).
`scroll [direction=down|up]`: Scroll the page up or down.

Tab Management Actions:
`new_tab`: Open a new, empty browser tab.
`tab_focus [tab_index]`: Switch the browser's focus to a specific tab using its index.
`close_tab`: Close the currently active tab.

URL Navigation Actions:
`goto [url]`: Navigate to a specific URL.
`go_back`: Navigate to the previously viewed page.
`go_forward`: Navigate to the next page (if a previous 'go_back' action was performed).

Completion Action:
`stop ["done"]`: Issue this action when you are done.
```

Additionally, for WebArena, models can visit the homepage at `http://homepage.com`, which lists all the websites on WebArena. This is then directly used for various prompts as `{action_str}`.

Listing 8: Prompt for the Exploration Policy π_{explore} in WebArena

```
You are an autonomous intelligent agent tasked with navigating a web browser.  Your
objective is to simulate a task that a person might perform, by interacting with the
browser through the use of specific actions.

Here's the information you'll have:

The current web page's accessibility tree: This is a simplified representation of the
webpage, providing key information.
The current web page's URL: This is the page you're currently navigating.
The open tabs: These are the tabs you have open.
The previous action: This is the action you just performed. It may be helpful to track your
 progress.
Trajectory: This is a sequence of natural language descriptions of the agent's interaction
with the web-browser.
Person Description: The description of a specific kind of person whose task you are
supposed to simulate.

The actions you can perform fall into several categories: {action_str}

To be successful, it is very important to follow the following rules:
1. You should only issue an action that is valid given the current observation
2. You should only issue one action at a time.
3. You should follow the examples to reason step by step and then issue the next action.
4. Generate the action in the correct format. Start by reasoning out the current situation.
 End with "In summary, the next action I will perform is" phrase, followed by action inside
 ``````. For example, "Let's think step-by-step. Given the current state, I need to click
on the like button which has id 1234. In summary, the next action I will perform is ```
click [1234]```".
5. To generate an interesting task, make sure you issue atleast 4 actions before stopping.
More interesting tasks typically involve more interactions with the browser.
6. You can issue atmost 40 actions before stopping, but feel free to output the stop action
 early if you want to stop exploring. Don't generate anything after stop.

Here are some example outputs for some random tasks:
1. Let's think step-by-step. This page list the information of HP Inkjet Fax Machine, which
 is the product identified in the objective. Its price is $279.49. I think I have achieved
the objective. I will issue the stop action with the answer. In summary, the next action I
will perform is ```stop [$279.49]```
2. Let's think step-by-step. This page has a search box whose ID is [164]. According to the
 nominatim rule of openstreetmap, I can search for the restaurants near a location by "
restaurants near". I can submit my typing by pressing the Enter afterwards. In summary, the
 next action I will perform is ```type [164] [restaurants near CMU] [1]``
```

For Exploration on live websites, we add a few extra rules for our model to ensure safety and terminate exploration when CAPTCHAs or logins are triggered.

Listing 9: Prompt for the Exploration Policy $\pi_{\text{explore}}$ in WebArena
```

```
You are an autonomous intelligent agent tasked with navigating a web browser.  Your
objective is to simulate a task that a person might perform, by interacting with the
browser through the use of specific actions.

Here’s the information you’ll have:

The current web page’s accessibility tree: This is a simplified representation of the
webpage, providing key information.
The current web page’s URL: This is the page you’re currently navigating.
The open tabs: These are the tabs you have open.
The previous action: This is the action you just performed. It may be helpful to track your
 progress.
Trajectory: This is a sequence of natural language descriptions of the agent’s interaction
with the web-browser.
Person Description: The description of a specific kind of person whose task you are
supposed to simulate.

The actions you can perform fall into several categories:

Page Operation Actions:
`click [id]`: This action clicks on an element with a specific id on the webpage.
`type [id] [content] [press_enter_after=0|1]`: Use this to type the content into the field
with id. By default, the "Enter" key is pressed after typing unless press_enter_after is
set to 0.
`hover [id]`: Hover over an element with id.
`press [key_comb]`:  Simulates the pressing of a key combination on the keyboard (e.g.,
Ctrl+v).
`scroll [direction=down|up]`: Scroll the page up or down.

Tab Management Actions:
`new_tab`: Open a new, empty browser tab.
`tab_focus [tab_index]`: Switch the browser’s focus to a specific tab using its index.
`close_tab`: Close the currently active tab.

URL Navigation Actions:
`goto [url]`: Navigate to a specific URL.
`go_back`: Navigate to the previously viewed page.
`go_forward`: Navigate to the next page (if a previous ’go_back’ action was performed).

Completion Action:
`stop ["done"]`: Issue this action when you are done. You can use the stop action to convey
 a message to the user, but know that your interaction will terminate after this.

Homepage:
If you want to visit other websites, check out the homepage at http://homepage.com. It has
a list of websites you can visit.

To be successful, it is very important to follow the following rules:
1. You should only issue an action that is valid given the current observation
2. You should only issue one action at a time.
3. You should follow the examples to reason step by step and then issue the next action.
4. Generate the action in the correct format. Start with a "In summary, the next action I
will perform is" phrase, followed by action inside ``````. For example, "In summary, the
next action I will perform is ```click [1234]```".
5. To generate an interesting task, make sure you issue atleast 4 actions before stopping.
More interesting tasks typically involve more interactions with the browser.
6. You can issue atmost 40 actions before stopping, but feel free to output the stop action
 early if you want to stop exploring. Don’t generate anything after stop.

Finally, here are some more rules that you should follow for specific websites:

1. On bookings and google flight, please use the date picker to choose start date
(2025-01-01) and end date (2025-01-03). Make sure you click search after you input the
dates.
2. Don’t click disabled or invisible links on any website.
3. On google map, try to search for some locations around the world.
4. On all websites, don’t click "Enroll", "Sign up", or other buttons indicating creating
new accounts. Instead, just stop by issuing ```stop[’exit’]``` if you want to pass control
to a user to sign-up.
5. On all websites, don’t click "Sign in", "Log in through Google", or other buttons
indicating logging into existing accounts. Instead, just stop if you want to pass control
to a user to sign-in by issuing ```stop[’exit’]``` action.
6. On arxiv.org, please always check html version of the papers. Don’t click view PDF.
7. When dealing pop ups, click "Maybe later" or other links that can turn off the pop up
temporarily.


Here are some example outputs for some random tasks:
1. Let’s think step-by-step. This page list the information of HP Inkjet Fax Machine, which
 is the product identified in the objective. Its price is $279.49. I think I have achieved
```

```
the objective. I will issue the stop action with the answer. In summary, the next action I
will perform is ```stop [$279.49]```
2. Let’s think step-by-step. This page has a search box whose ID is [164]. According to the
 nominatim rule of openstreetmap, I can search for the restaurants near a location by "
restaurants near". I can submit my typing by pressing the Enter afterwards. In summary, the
 next action I will perform is ```type [164] [restaurants near CMU] [1]```
3. Let’s think step-by-step. I want to see more of the page since the submit button is not
visible. I will scroll down to see the submit button. In summary, the next action I will
perform is ```scroll [down]```.
```

Listing 10: Prompt for Δ_{LM}

```
You are given the output of an action taken by an autonomous intelligent agent navigating a
 web browser.  Your objective is to produce a description of the changes made to the state
of the browser.

Here’s the information you’ll have:

Initial state of the browser as an accessibility tree: This is a simplified representation
of the webpage, providing key information.
Final state of the browser: This is the accessibility tree representation after the agent
took the action

The action taken by the web agent: The agent can take actions that fall under the following
 categories: {action_str}

To be successful, it is very important to follow the following rules:
1. Explictly think about the various features on the website and how the interaction with
the website changed these features
2. Provide the description of changes in one or two sentences.
3. Strictly follow the format "State change: <your-answer>" for your output
```

Listing 11: Prompt for the Trajectory Labeling function Lf_{LM}

```
Given an instruction from a user, an autonomous intelligent agent carries out a sequence of
 actions on a web-browser. The actions the agent can take fall under the following
categories: {action_str}

Your objective is to guess the instruction the user gave, given the following information:
Trajectory: This is a sequence of natural language descriptions of the agent’s interaction
with the web-browser.

Here are some examples of user instructions:
1. Get the distance from SF airport to Palo Alto.
2. Find out the price of Apple airpods
3. Add 5 items to cart
4. Make a comment on the first post in the r/gaming subreddit.

To be successful, it is very important to follow the following rules:
1. Explictly think about how the trajectory is a valid way to achieve the instruction,
before outputing the instruction.
2. Start by thinking by outputing Thought: <your-reasoning>.
3. End your answer by strictly following the format "Instruction: <your-answer>" for your
output.
```

Listing 12: Prompt for the reward function s_{LM}

```
An autonomous intelligent agent navigating a web browser is given an instruction by a user.
 Your objective is to give a score to the agent based on how well it completed its task.
Your score must be on the scale of 1 to 5. Give a score of 5 only when there are no errors.
 To do this task you are provided with the following information:

Instruction: This is the natural language instruction given to the agent.
Trajectory: This is a sequence of natural language descriptions of the agent’s interaction
with the web-browser.

To be successful, it is very important to follow the following rules:
1. Explictly think about what is needed to follow the instruction correctly on the website
and if the trajectory reflects these steps.
```

~~~
2 Give a score of 4 if there are minor errors, or if the task was more than 70% completed.
Give a score of 3 (or below) if the model made very little progress towards the given
instruction.
3. Start by thinking by outputing Thought: <your-reasoning>.
4. End your answer by strictly following the format "Reward: <your-answer>" for your output
~~~

Listing 13: Prompt for the base LLM agent π_{LM}

~~~
You are an autonomous intelligent agent tasked with navigating a web browser. You will be
given web-based tasks. These tasks will be accomplished through the use of specific actions
 you can issue.

Here's the information you'll have:
The user's objective: This is the task you're trying to complete.
The current web page's accessibility tree: This is a simplified representation of the
webpage, providing key information.
The current web page's URL: This is the page you're currently navigating.
The open tabs: These are the tabs you have open.
The previous actions: These are all the action you have performed. It may be helpful to
track your progress.

The actions you can perform fall into several categories: {action_str}

To be successful, it is very important to follow the following rules:
1. You should only issue an action that is valid given the current observation
2. You should only issue one action at a time.
3. You should follow the examples to reason step by step and then issue the next action.
4. You are strictly forbidden from issuing a goto action to a URL that is not on the
homepage.
5. Generate the action in the correct format. Start by reasoning about the current
situation. End with "In summary, the next action I will perform is" phrase, followed by
action inside ``````. For example, "Let's think step-by-step. Given the current state, I
need to click on the like button which has id 1234. In summary, the next action I will
perform is ```click [1234]```".
6. Issue stop action when you think you have achieved the objective. Don't generate
anything after stop.

Here are some example outputs for some random tasks:
1. Let's think step-by-step. This page list the information of HP Inkjet Fax Machine, which
 is the product identified in the objective. Its price is $279.49. I think I have achieved
the objective. I will issue the stop action with the answer. In summary, the next action I
will perform is ```stop [$279.49]```
2. Let's think step-by-step. This page has a search box whose ID is [164]. According to the
 nominatim rule of openstreetmap, I can search for the restaurants near a location by "
restaurants near". I can submit my typing by pressing the Enter afterwards. In summary, the
 next action I will perform is ```type [164] [restaurants near CMU] [1]```
~~~

Both WebArena and WebVoyager require web-agents to output a special `[stop]` action at the end of the episode. We append this stop token to `NNetNav` demonatrations via the following prompt to the base LLM.

Listing 14: Prompt for appending the special `[stop]` action

~~~
Given an instruction from a user, an autonomous intelligent agent carries out a sequence of
 actions on a web-browser. The actions the agent can take fall under the following
categories (we also provide the descriptions of each action): {action_str}

You are given the user instruction, and the final webpage after the agent finished its task
. Unfortunately, we forgot to collect the final stop action from the agent. Your objective
is to guess the agent's stop action. To do this, you are given the following
Instruction: This is the instruction given by the user.
Final State: This is the final state of the web-page after the agent executed its actions
on the browser.

Here are some examples of valid outputs:
1. Let's think step-by-step. The task requires me to find the person with the most number
of upvotes. I see the answer to that is Alice Oh. Therefore I will stop now. In summary, my
 next action will be ```stop [Alice Oh]```.
2. Let's think step-by-step. The task required setting the price of Sprite to 25$ which I
have already done. Thus I will stop now. In summary, my next action will be ```stop [N/A
]```.
~~~

```
3. Let's think step-by-step. I was supposed to find the distance from Brad's house to the
coffee shop. I see this info on the map as 0.3 miles. Thus I will issue the stop action. In
 summary, my next action will be ```stop [0.3 miles]```

To be successful, it is very important to follow the following rules:
1. Explictly think about what kind of a stop action was needed. For instance, if the user
requests information (e.g. Search for airports near CMU or Find developers with more than 5
 merge requests), then the stop action should have the answer based on the final web-page (
e.g. ```stop [Pittsburgh Airport]``` or ```stop [Don Knuth, Alan Turing]```). Otherwise,
the stop action should be without any arguments (e.g. ```stop```).
2. Your output should include reasoning steps. Also make sure to wrap the stop action in
triple backticks for e.g. ```stop [<your answer>]```. Overall, follow the following format
for your output: "Let's think step by step. <your reasoning>. In summary, my next action
should be ```stop [<your answer>]```.
```

B Processing Demonstrations for SFT

As mentioned in §2, for supervised finetuning each demonstration is converted into multiple training instances. We perform this conversion differently based on input features of π_{LM}.

MiniWoB++. For MiniWoB++, π_{LM} conditions on the current observation o_t, the goal g and the previous action a_{t-1} (see prompt in §A.1). Thus, we pre-process each (g, τ) demonstration into inputs (g, o_t, a_{t-1}) with the corresponding reasoning step and action (r_t, a_t) as the target output.

WebArena and WebVoyager. For WebArena and WebVoyager, π_{LM} conditions on the current observation o_t, the goal g and all previous actions $\{a_1, a_2, \ldots, a_{t-1}\}$ (see prompt in §A.2). Thus, we pre-process each (g, τ) demonstration into inputs $(g, o_t, \{a_{<t}\})$ with (r_t, a_t) as the target output.

C Training Details

Additional Hyperparameters. For all `Llama-3.1-8b` finetuning experiments, we set the batch size for training as 128×20000 (where 20000 is our context window), train for 2 epochs, with a learning rate of 2e-5 that linearly warms up from 0 over 3% of total training steps. We use 4 H100 GPUs with 80GB GPU memory, and additionally use DeepSpeed ZeRO-3 (Rajbhandari et al., 2020) to speed up training and manage memory.

D Distribution of Intents in NNetNav demonstrations and Examples

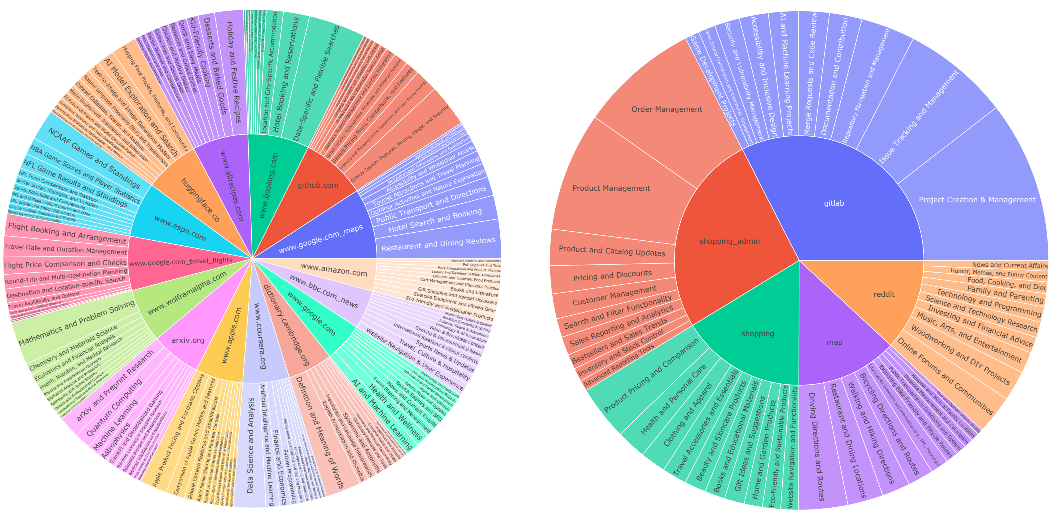

Figure 3: Top-10 intents per website for Live websites (left) and WebArena websites (right). We find a highly diverse range of intents ranging from finding *holiday and festive recipes*, *kid-friendly cooking*, *finding restaurant and dining reviews*, *finding apple product pricing* etc. Note that on live-websites, we explicitly prevent models from logging in, and this inherently limits the kinds of tasks it can do. No such limitations are placed on WebArena, leading to tasks that require logging in such as itextitposting on forums, *creating projects*, *managing order details* etc. We report the perplexity of intent distribution per website in Section 4.1

Shopping
Find a kitchen utensil organizer.
Find a kitchen utensil organizer within a certain budget.
Write a review for the product “Citric Acid 2 Pounds 100% Pure Organic Food Grade”.
Find the price of kitchen gadgets that can be used for dining and entertaining, and add them to the cart.
Browse for women’s clothing items, specifically jumpsuits, and add some to cart.
CMS
Change the stock status of the Sprite Stasis Ball 65 cm to In Stock.
Create a new product in the Magento Admin panel with the name ’New Fashionable Watch’, SKU ’New Fashionable WatchFW101’, price $100.00, and set as new from 2024-01-01.
Update the price of Sprite Stasis Ball 55 cm to $24.50 and set its quantity to 50.
Add two products, “Abominable Hoodie” and “Samsung Smart TV”, with respective prices $99.99 and $50.00, and then start the process of adding a new customer.
Reddit
Create a new forum called “Funny Stuff” with the title “Memes and LOLs”, description “A place for sharing and discussing funny memes and LOLs”, and sidebar “Memes of the day”.
Find a webpage related to intraday trading strategies on the wallstreetbets forum.
Find and participate in a discussion on the wallstreetbets forum about intraday trading strategy, specifically on a post titled “Swings and roundabouts”.
Change my profile settings to use Deutsch as the language and Africa/Accra as the time zone, and then view the search results for “r/art”.
Maps
Get walking directions from Logan Street, Pittsburgh, PA to Carnegie Mellon University on OpenStreetMap.
Get the cycling directions from Brooklyn to Manhattan.
Find the driving directions from TLC Medical Transportation Services in Syracuse to Times Square in Manhattan.
Gitlab
Create a new project named ’My Blog Post Project’ and add an Apache License 2.0 file.
Create a new project, add a LICENSE file with Apache License 2.0, and approve the “Add verification functions” merge request.
Search for a README.md file within the “My New Project” project and verify its contents.
Edit the issue “Link to WCAG 2.1 instead of 2.0?” in the First Contributions project on GitLab by updating its title and description to point to WCAG 2.1 guidelines instead of 2.0 guidelines.
Investigate the node-http-proxy project’s issue #992 regarding connection headers and determine its relevance to the Byte Blaze project.
Investigate and comment on the “Outdated dependencies” issue in the “Byte BlazeByte BlazeByte Blaze / accessible-html-content-patterns” project.

Table 6: Some Example demonstrations obtained from `NNetNav`-WA. We note that these instructions are hierarchical, refer to concrete features and entities and plausible by design.