

Abstract

AI agents are an exciting new research direction, and agent development is driven by benchmarks. Our analysis of current agent benchmarks and evaluation practices reveals several shortcomings that hinder their usefulness in real-world applications. First, there is a narrow focus on accuracy without attention to other metrics. As a result, SOTA agents are needlessly complex and costly, and the community has reached mistaken conclusions about the sources of accuracy gains. Our focus on cost in addition to accuracy motivates the new goal of jointly optimizing the two metrics. We design and implement one such optimization, showing its potential to greatly reduce cost while maintaining accuracy. Second, the benchmarking needs of model and downstream developers have been conflated, making it hard to identify which agent would be best suited for a particular application. Third, many agent benchmarks have inadequate holdout sets, and sometimes none at all. This has led to agents that are fragile because they take shortcuts and overfit to the benchmark in various ways. We prescribe a principled framework for avoiding overfitting. Finally, there is a lack of standardization in evaluation practices, leading to a pervasive lack of reproducibility. We hope that the steps we introduce for addressing these shortcomings will cause the development of agents that are useful in the real world and not just accurate on benchmarks.

1 Introduction

Compound AI systems, or AI agents, are becoming an important research direction. Zaharia et al. [63] argue that “compound AI systems will likely be the best way to maximize AI results in the future, and might be one of the most impactful trends in AI in 2024.” Over a dozen agent benchmarks have been released, spanning domains such as web interaction [66], programming [21] and tool use [39]. Many benchmarks developed for LLM evaluation have also been used for agent evaluation.

Agent evaluation differs from language model evaluation in fundamental ways. Agents can be used on tasks that are harder, more realistic, have more real-world utility, and usually don’t have a single correct answer. For example, agents can use the common line to carry out tasks: `SWEE-agent` even includes its own agent code repository [59]. Agents can cost much more than a single model call. For example, the authors of `SWEE-agent` cap each run of the agent at \$4 USD, which translates to hundreds of thousands of language model tokens.

As a result, agent benchmarking comes with distinct challenges. This paper systematically demonstrates these challenges and provides recommendations for addressing them. Specifically, we make five contributions.

1. AI agent evaluations must be cost-controlled (Section 2). The language models underlying most AI agents are stochastic. This means simply calling the underlying model multiple times can increase accuracy [27, 6, 26]. We introduce three simple baseline agents and empirically

*Equal Contribution. Contact: {sanyas, strobs}@princeton.edu

show that they outperform many SOTA complex agent architectures on HumanEval [64, 65, 64] while costing much less. Therefore, agent evaluation must be cost-controlled; otherwise it will encourage researchers to develop extremely costly agents just to claim they topped the leaderboard.

2. Jointly optimizing accuracy and cost can yield better agent design (Section 3). Visualizing evaluation results as a Pareto curve of accuracy and inference cost opens up a new space of agent design: jointly optimizing the two metrics. We modify the DSPy framework [24] for joint optimization, lowering cost while maintaining accuracy on HumanEval [66].

3. Model developers and downstream developers have distinct benchmarking needs (Section 4). Through a case study of `NovoQA` [53], we show how benchmarks meant for model evaluation can be misleading when used for downstream evaluation. We argue that downstream evaluation should account for dollar costs, rather than proxies for cost such as the number of model parameters.

4. Agent benchmarks enable shortcuts (Section 5). We show that many types of overfitting to agent benchmarks are possible. We identify four levels of generality of agents and argue that different types of hold-out samples are needed based on the desired level of generality. Without proper hold-outs, agent developers can take shortcuts, even unintentionally. We illustrate this with a case study of the `WebArena` benchmark [66].

5. Agent evaluation lacks standardization and reproducibility (Section 6). We found pervasive shortcomings in the reproducibility of `WebArena` and HumanEval evaluations [24, 66]. These errors inflate accuracy estimates and lead to overoptimism about agent capabilities.

The overarching goal of our work is to stimulate the development of agents that are useful in the real world and not just accurate on benchmarks. (1) and (2) above show how the variable cost of running an agent impacts the results of evaluation. (3) and (4) and (5) do so by improving precision about what a benchmark aims to measure and ensuring that it actually measures that; and (3) and (4) do both.

1.1 What is an AI agent?

In traditional AI, agents are defined as entities that perceive and act upon their environment [40]. In the LLM era, the term is used in a narrower way: a thermostat would qualify as an agent under the traditional definition. Many researchers have tried to formalize the community’s intuitive understanding of what constitutes an agent in the context of language-model-based systems. Many of them view it as a spectrum — sometimes defined by the term “*agent*” [38] — rather than a binary definition of an agent. We agree with this perspective. Since there are already many definitions, we do not provide a new one, but rather identify the factors that cause an AI system to be considered more agent-like according to existing definitions. We found three clusters of factors.

• Environment and goals. The more complex the environment — e.g., range of tasks and domains, multi-stakeholder, long time horizons, unexpected changes — the more AI systems operating in that environment are agent-like [41, 14]. Systems that parse complex goals without being instructed on how to pursue the goal are more agent-like [41, 14].

• User interface and supervision. AI systems that can be instructed in natural language and act autonomously on the user’s behalf are more agent-like [41]. In particular, systems that require less user supervision are more agent-like [41, 4, 14]. We discuss the user supervision aspect in more detail in Section 5.2.

• System design. Systems that use design patterns such as tool use (e.g., web scraping, programming) or planning (e.g., reflection, radical decomposition) are more agent-like [57, 38]. Systems whose control flow is driven by an LLM, and hence dynamic, are more agent-like [57, 5].

2 AI agent evaluations must be cost-controlled

2.1 Maximizing accuracy can lead to unbounded cost

Calling language models repeatedly and taking a majority vote can lead to non-trivial increases in accuracy across benchmarks like GSM-8K, MATH, Chess, and MMLU [26, 4, 68].

When the agent environment has easy signals to check if an answer is correct, repeatedly retrying can lead to even more compelling performance gains [51]. Li et al. [27] showed that the accuracy

joint optimization allows us to trade off the fixed and variable costs of running an agent. By spending more upfront on the one-time optimization of agent design, we can reduce the variable cost of running an agent (e.g., by finding shorter prompts and few-shot examples while maintaining accuracy).

As an illustration of the potential of joint optimization, we modify the DSPy framework [24] and evaluate it on the HumanEval benchmark [66]. We chose `HumanEval` because it is one of the benchmarks used to illustrate the effectiveness of DSPy in the original paper and has been featured in several official tutorials by the developers. We use the OpenAI hyperparameter optimization framework [2] to search for few-shot examples to be included with an agent that maintains cost while maintaining accuracy. Note that we expect more complex joint optimization approaches to vastly outperform our approach. Our results are only a starting point intended to illustrate the vast, underexplored design space in agent design enabled by joint optimization.

3.1 HumanEval evaluation setup

We implement several agent designs to evaluate performance on multi-hop question-answering using DSPy. For retrieval, we use `ColBERTv2` to query Wikipedia based on the HumanEval tasks specification [60]. Performance is evaluated by comparing whether the accuracy successfully recalled all ground-truth documents that are part of the specific task. We use 100 samples from the HumanEval training set to optimize the DSPy pipelines and 200 samples from the evaluation set to evaluate the results (this is consistent with the implementation of the DSPy pipelines provided by the developers to illustrate efficacy at multi-hop retrieval). We evaluate five agent architectures:

- Uncompiled:** We do not optimize the agent’s prompt or include instructions on formatting `HumanEval` queries. Each prompt only contains the instructions for the task and the main context (i.e., question, context, reasoning) but no few-shot examples or formatting instructions.
- Formatting instructions only:** This is the same as the uncompiled baseline, but we add instructions on how to format generated outputs for writing retrieval queries.
- Few-shot:** We use DSPy to identify effective few-shot examples using all 100 samples from the training set. We include formatting instructions. Few-shot examples are selected based on successful predictions generated on the training set.
- Random search:** We use DSPy’s random search optimizer on a subset of the training data (50 of 100 samples) to select the best few-shot examples based on its performance on the remaining 50 samples. We include formatting instructions.
- Joint optimization:** We iterate over half the training set (50 of 100 samples) to collect a set of candidate few-shot examples that improve the model’s accuracy. We use the other 50 samples for validation. We jointly maximize accuracy and minimize the number of tokens in the few-shot examples. We use the first 50 samples from the training set to optimize the DSPy pipelines and 200 samples from the evaluation set to evaluate the results. We search over the following parameters to find Pareto-optimal agent designs: (a) the temperature for each module within the agent, (b) the number of few-shot examples, (c) the selection of specific examples, and (d) whether to add formatting instructions. Of the candidate agents selected by Optuna, we pick the one with the best accuracy on the development set as our joint optimization model.

We test all five of the above agent designs on two underlying models: `Llama-3.70B` and `GPT-3.5`.

3.2 HumanEval results: Joint optimization reduces cost while maintaining accuracy

Fig. 2 shows our main results. We confirm that DSPy offers substantial accuracy improvements over uncompiled baselines, but we find that this comes at a cost. Fortunately, we can mitigate the cost overhead — for `GPT-3.5`, joint optimization leads to 53% lower variable cost with similar accuracy compared to both default DSPy implementations. Similarly, for `Llama-3.70B`, it leads to a 41% lower cost while maintaining accuracy.

Tradeoffs between fixed and variable costs for agent design. Our joint optimization framework provides a way to trade off fixed and variable costs (Appendix B). In particular, we find that if we use HumanEval tasks, the `Llama-3.70B` as well as the `GPT-3.5` joint optimization model both become cheaper (in terms of total cost) compared to the default DSPy implementation, after 1,350 tasks or more.

Tradeoffs between fixed and variable costs for agent design. Our joint optimization framework provides a way to trade off fixed and variable costs (Appendix B). In particular, we find that if we use HumanEval tasks, the `Llama-3.70B` as well as the `GPT-3.5` joint optimization model both become cheaper (in terms of total cost) compared to the default DSPy implementation, after 1,350 tasks or more.

Tradeoffs between fixed and variable costs for agent design. Our joint optimization framework provides a way to trade off fixed and variable costs (Appendix B). In particular, we find that if we use HumanEval tasks, the `Llama-3.70B` as well as the `GPT-3.5` joint optimization model both become cheaper (in terms of total cost) compared to the default DSPy implementation, after 1,350 tasks or more.

Tradeoffs between fixed and variable costs for agent design. Our joint optimization framework provides a way to trade off fixed and variable costs (Appendix B). In particular, we find that if we use HumanEval tasks, the `Llama-3.70B` as well as the `GPT-3.5` joint optimization model both become cheaper (in terms of total cost) compared to the default DSPy implementation, after 1,350 tasks or more.

Tradeoffs between fixed and variable costs for agent design. Our joint optimization framework provides a way to trade off fixed and variable costs (Appendix B). In particular, we find that if we use HumanEval tasks, the `Llama-3.70B` as well as the `GPT-3.5` joint optimization model both become cheaper (in terms of total cost) compared to the default DSPy implementation, after 1,350 tasks or more.

This depends entirely on the agent’s purpose and the benchmark creator designs in terms of the generality of the agent [10, 35]. In our survey, we have found four levels of generality:

- 1. Distribution-specific benchmarks** are limited to a specific task, such as U.S. grade school math problems, and do not account for distribution shifts or generalization.
- 2. Task-specific benchmarks** are limited to a specific task such as bookkeeping, playing an order on an e-commerce website [61], or solving a GitHub issue [21], and account for the possibility of distribution shifts, including drift. After all, an agent that can book flights today but breaks if the flight booking website changes its layout would not be very useful.
- 3. Domain-general benchmarks** aim to measure the ability to perform any task in a specific domain, such as web browsing or tool use [66].
- 4. General-purpose benchmarks** measure the accuracy of agents across different domains, such as the same agent being able to perform web browsing and robotics tasks. It is unclear if such benchmarks are necessary or if aggregating domain-general benchmarks can better serve the purpose.

We propose as a core principle that the more general the intended generality of the agent, the more it should set aside drift from the training set, as detailed in Table 1. For example, if a benchmark is intended to be domain-general but doesn’t contain hold-out tasks, agent developers may (deliberately or unintentionally) take shortcuts that work only for the specific tasks represented in the dataset, resulting in agents that don’t work well for other tasks. We argue that benchmarks that create a better test to ensure that shortcuts are impossible. We view this as the responsibility of benchmark developers rather than agent developers, because designing benchmarks that do not allow shortcuts is much easier than checking every single agent to see if it takes shortcuts. Benchmarks that create a level playing field are the core reason for the rapid progress of ML over the last half century [12, 46].

Level of generality	What should be held out	Num. benchmarks with appropriate holdouts
Distribution-specific	In-distribution samples	1 / 6
Task-specific	Out-of-distribution samples	1 / 6
Domain-general	Tasks	1 / 8
Fully general	Domains	2 / 2

Table 1: Appropriate holdouts based on level of generality. See Appendix for full details.

There are many types of distribution shifts, and benchmark developers can’t necessarily model all of them. But they must attempt to identify which distribution shifts are particularly likely for the task in question. Another approach — not always practical — is to evaluate *in-situ* transfer, where leading agents are evaluated not just on benchmark tasks but also on the corresponding real-world tasks — for example, Amazon shopping for a web shopping benchmark [61].

We analyzed 17 agent benchmarks into the four levels of generality (Table A4). Most are either task-specific or domain-general. In many cases, it wasn’t clear which level of generality the benchmark developers had in mind, and we made our best guess based on how it was presented in the paper. This lack of clarity is problematic as it makes it hard to know what we can and can’t conclude about the agents that perform well on the benchmark.

We recognize that time and resource constraints may hinder benchmark designers from creating a holdout set at the correct generality level. Hence we count a holdout as appropriate if they actually create a holdout at the appropriate level of generality or if the benchmark designers indicate an intent to create such a holdout. However, as shown in Table 1 and Appendix C, the majority of benchmarks do not include an appropriate holdout set, including 7 that have no hold-out and do indicate that they intend to create such a holdout. We would like to see future editions of the benchmarks.

Note that in traditional machine learning research, hold-out test sets are usually at the level of individual out-of-distribution samples (the first two rows of Table 1). This is sufficient to ensure that models generalize to new tasks. For example, classifying images by species using LLMs and domain-general agents are expected to handle tasks that are not known ahead of time and may be specified in natural language in some cases. This motivates the need for hold-out tasks during evaluation.

of AlphaCode increases from close to 0% zero-shot to over 15% with 1,000 retries and over 30% with a million retries (accuracy is measured by how often one of the top 10 answers generated by the model is correct). Thus, there is seemingly no limit to the amount of inference compute that can increase accuracy, and increasing the generation budget has been shown to improve performance in various applications [56]. Coding competitions often include signals of correctness, such as test cases to check if a given solution is correct. Agent developers can keep sampling from an underlying model until the solution passes the test cases. Our results show that this is true for HumanEval.

2.2 Visualizing the accuracy-cost tradeoff using a Pareto curve

In the last year, many agents have been claimed to achieve state-of-the-art accuracy on coding tasks. But at what cost? To visualize the tradeoff, we re-evaluated the accuracy of three agents.

Specifically, we included agents from the HumanEval leaderboard on `PapersWithCode` that share their code publicly [7]: `LDB` [68], `LATS` [65], and `Reflexion` [44].¹ These agents rely on running the code generated by the model, and if it fails the test cases provided with the problem description, they try to debug the code [64]. Look at alternative paths in the code generation process [65], or “reflect” on why the model’s outputs were incorrect before generating another solution [44, 65, 64].

We also evaluated the cost and time requirements of running these agents. In addition, we calculated accuracy, cost, and running time of a few simple architectures:

• `Reflexion` and `GPT-4` models (zero-shot). Note that an agent to zero, up to five times, if it fails the test cases provided with the problem description. Retrying makes sense because LLMs aren’t deterministic even at temperature zero (Appendix A).

• `Warning`. This is the same as the retry strategy, but we gradually increase the temperature of the underlying model with each run, from 0 to 0.5. This increases the stochasticity of the model and, we hope, increases the likelihood that at least one of the retries will succeed.

• `Escalation`. We start with a cheap model (`Llama-3.5B`) and escalate to more expensive models (`GPT-3.5`, `Llama-3.70B`, `GPT-4`) if we encounter a test case failure.

We use the modified benchmark version of HumanEval with the LDB parser [64] since it includes example test cases for all 164 tasks (in the original benchmark, example test cases are provided for only 161 of 164 tasks, as detailed in Section 6).

2.3 Two-dimensional evaluation yields surprising insights

Fig. 1 shows our main results for this section. Note that an agent is on the Pareto frontier if there is no other agent that has significantly better performance on both dimensions simultaneously (see Appendix A.1).

“State-of-the-art” agent architectures for HumanEval do not perform simple baselines. There is no significant accuracy difference between our warning strategy and the best-performing agent architecture. In fact, we are not aware of any papers that use their proposed agent architectures with any of the last three of our simple baselines on HumanEval (retry, warning, escalation).²

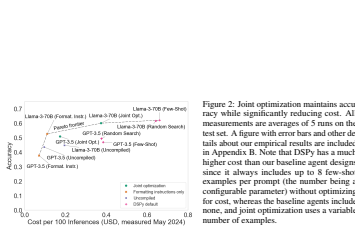
Agents differ drastically in terms of cost. For substantially similar accuracy, the cost can differ by almost an order of magnitude. Yet, the cost of running the agents for our proposed agent architectures in any of these papers, `Reflexion` and `LDB` cost over 50% more than the warning strategy, and `LATS` over 50 times more than all three costs are entirely or predominantly from calls to `GPT-4`, so these ratios

¹Reflexion is absent from the `PapersWithCode` leaderboard, but it has a reported accuracy of 91% (higher than any other agents with publicly available code apart from LDB and LATS), so we included it in our analysis. `AgentCode`, listed as the top-performing agent, did not include a link to the code on the benchmark at the time of our analysis (late April 2024), so we did not include it in our analysis. `Reflexion` and `LATS` are not on the frontier.

²We contacted the `PapersWithCode` to confirm, because given two points, a and b , on the graph corresponding to agents A and B , we can always linearly interpolate between them by creating a new agent that invokes A with probability p and B with probability $1 - p$. Hence, for instance, `agentCode` and `GPT-4` is not on the frontier.

³Chen et al. [6] do compare the effects of retrying the same model multiple times on HumanEval, but they do not test the performance of `GPT-4`, so we do not report for a specific complex agent architecture.

Figure 1: Joint optimization maintains accuracy while significantly reducing cost. All measurements are averages of 5 runs on the test set. Figure with error bars and other details about our empirical results are included in Appendix B. Note that DSPy has a much higher cost than our baseline agents, even since it always includes up to 8 few-shot examples per prompt (the number being a configurable parameter) without optimizing for cost, whereas the baseline agents include none, and joint optimization uses a variable number of examples.



(Appendix B). For agents used in large-scale real-world tasks, the variable cost is by far the dominant term compared to the fixed cost, as an order of magnitude, as an agent might be used for millions of times. In summary, joint optimization allows for efficient agent design. This comes at a small fixed cost for optimizing the design, which is insignificant if the agent is used thousands of times.

4 Model and downstream developers have distinct benchmarking needs

AI evaluations are used by model developers and AI researchers to identify which changes to the training data and architecture improve accuracy (model evaluation) and also by downstream developers to decide which AI systems to use in their products for downstream decision-making. The difference between model evaluation and downstream evaluation is underappreciated. This has led to much confusion about how to factor in the cost of running AI.

Model evaluation is a scientific question of interest to researchers. Here, it makes sense to stay away from dollar costs, because reporting costs breaks many properties of benchmarks that we take for granted: measurements don’t change over time (whereas costs tend to come down) and different models compete on a level playing field (whereas some developers may benefit from economies of scale, leading to lower inference costs). Because of this, researchers usually pick a different axis for the Pareto curve, such as the amount of compute or the amount of model parameters.

For model evaluation, controlling for compute is a reasonable approach: if we normalize the amount of compute used to train a model, we can then understand if factors like architectural changes or changes in the data can improve performance for improvements in compute [25].

Downstream evaluation is an engineering question that helps inform a practical decision of which model or agent to use in a particular application. Here, it’s the actual cost of deployment. The downsides of cost measurement in model evaluation are exactly what we need for downstream evaluation. Namely, inference costs do come down over time, and that greatly matters to downstream developers. It is unnecessary and counterproductive for the evaluation to stay frozen in time.

Proxies for cost are misleading for downstream evaluation. In the context of downstream evaluation, proxies for cost (such as the number of parameters or amount of compute used) are misleading. For example, Mistral released a figure alongside their latest model, Mistral 8x22B, to explain why developers should choose it over competitors [47]. It used the number of active parameters as a proxy for cost. From the perspective of a downstream developer, this proxy is misleading. For example, as of June 2024, Mistral 8x7B costs twice as much as `Llama-3.1B` on open-source providers. Any such proxy shows that it costs about the same because it doesn’t consider the number of active parameters.

Downstream developers don’t care about the number of active parameters when they’re using an API. They simply need to know the dollar cost of inference and the accuracy of the active parameters used in the application. Mistral’s use of “active parameters” as a proxy for cost is misleading because it makes their models look better than dense models such as Meta’s `Llama` and Cohere’s `Command R+`. If every model developer picked a proxy that makes their model look good, multi-dimensional evaluation would lose its usefulness.

Downstream developers don’t care about the number of active parameters when they’re using an API. They simply need to know the dollar cost of inference and the accuracy of the active parameters used in the application. Mistral’s use of “active parameters” as a proxy for cost is misleading because it makes their models look better than dense models such as Meta’s `Llama` and Cohere’s `Command R+`. If every model developer picked a proxy that makes their model look good, multi-dimensional evaluation would lose its usefulness.

Downstream developers don’t care about the number of active parameters when they’re using an API. They simply need to know the dollar cost of inference and the accuracy of the active parameters used in the application. Mistral’s use of “active parameters” as a proxy for cost is misleading because it makes their models look better than dense models such as Meta’s `Llama` and Cohere’s `Command R+`. If every model developer picked a proxy that makes their model look good, multi-dimensional evaluation would lose its usefulness.

Downstream developers don’t care about the number of active parameters when they’re using an API. They simply need to know the dollar cost of inference and the accuracy of the active parameters used in the application. Mistral’s use of “active parameters” as a proxy for cost is misleading because it makes their models look better than dense models such as Meta’s `Llama` and Cohere’s `Command R+`. If every model developer picked a proxy that makes their model look good, multi-dimensional evaluation would lose its usefulness.

Downstream developers don’t care about the number of active parameters when they’re using an API. They simply need to know the dollar cost of inference and the accuracy of the active parameters used in the application. Mistral’s use of “active parameters” as a proxy for cost is misleading because it makes their models look better than dense models such as Meta’s `Llama` and Cohere’s `Command R+`. If every model developer picked a proxy that makes their model look good, multi-dimensional evaluation would lose its usefulness.

Downstream developers don’t care about the number of active parameters when they’re using an API. They simply need to know the dollar cost of inference and the accuracy of the active parameters used in the application. Mistral’s use of “active parameters” as a proxy for cost is misleading because it makes their models look better than dense models such as Meta’s `Llama` and Cohere’s `Command R+`. If every model developer picked a proxy that makes their model look good, multi-dimensional evaluation would lose its usefulness.

Web agents can be evaluated on many capabilities: navigating to a website, scrolling, selecting the right web element etc. There are many different types of websites that can be used such as benchmarks: e-commerce, social media, informational search etc. `WebArena` is an agent benchmark that aims to evaluate agents on tasks on the web [66]. It includes clones of six different websites (GitHub, LinkedIn, Wikipedia, OpenStreetMap, and a calculator) and a custom natural language interface with two calculators (calculator and scriptspicy). It has 812 different tasks that involve interacting with these websites, such as “Find the address of all US international airports that have within a driving distance of 60 km to the Niagara Falls” and “post a question on a subreddit related to New York City”.

`WebArena`’s core stated selling point seems to be realism, which means that it should be difficult to find shortcuts. If agents should be robust to changes made to a website over time. However, `WebArena` does not model drift.

To be clear, it is challenging to model drift. Benchmark developers would need to find changes made to published websites and incorporate them into the training set. But even if they did, the hold-out set would need to be kept secret, since agent developers could otherwise overfit to the specific changes in the benchmark. Still, we view these steps as necessary for meaningful evaluation. Consider the top agent on the `WebArena` leaderboard, called `STEP` [47]. It has an accuracy of 35.8%, more than double the accuracy of the top-performing baseline agent introduced in the `WebArena` paper, and over 10 percentage points more than the next-best agent [13]. How does `STEP` achieve this high accuracy?

It turns out that `STEP` hardcodes policies to solve the specific tasks included in `WebArena`. For example, several `WebArena` Reddit tasks involve navigating to a user’s profile. The `STEP` policy for this task is to look at the current base URL, and add a suffix `/user/user_name/`. This is brittle: the policy would no longer be effective if the website updates its URL structure (an example of drift). Even if the probability of an individual policy failing is small, an agent might need to call different policies dozens of times for each task. The overall probability of failure compounds quickly.

To be clear, the `STEP` developers’ goals are orthogonal to the benchmark developers’ goals — creating composable policies for accomplishing fixed tasks that are known a priori. From this perspective, `STEP`’s design choices make sense.

Yet the leaderboard accuracy on `WebArena` (such as the accuracy of the `STEP` agent) is misleading from the perspective of downstream developers, who might be using the `WebArena` leaderboard to understand the accuracy of web agents on real-world tasks and make decisions about which agent to adopt in an application.

Things become even more problematic if we consider `WebArena` a domain-general benchmark. This can be justified based on the claim that for previous web benchmarks, “the functions that most environments is a limited version of their real-world counterparts, leading to a lack of task diversity” [66]. This suggests that the `WebArena` developers aim to simulate the development of general-purpose web agents on real-world tasks and not on specific tasks on the web.

Unfortunately, `WebArena` lacks a hold-out test set for evaluating whether an agent can perform well on unseen web tasks. It is hard to confirm if building a domain-general benchmark is indeed their objective. This lack of clarity is problematic as it makes it hard to assess the utility of the benchmark.

Notice that if the hold-out set contained different tasks (such as samples from completely new and unseen websites) that were not in the training set, the accuracy of agents like `STEP` would be drastically lower, because none of the hardcoded policies would be effective.

4.2 Agent benchmarks don’t account for humans in the loop

The degree of human supervision, feedback, and intervention required for an agent to perform a task can be seen as a spectrum. Consider a data analysis task. On one end of the spectrum, the analyst might use a chart to help with tasks like debugging. Here the user is firmly in control and verifies all charted outputs. Or the analyst might ask the agent to write and execute code for certain data

¹See <https://hmsk1stxtr0b.github.io/agent-eval-website/>.

²We can go further. What about agents that are not using Python programming language? For example, if an agent chooses to program in a Python programming language but cannot program in any other language, is it highly

