Bootstrapping LLM Agents via Verification

Anonymous ACL submission

Abstract

We present a self-training method that allows language model-based agents to improve performance without distilling proprietary models. Existing self-verification methods struggle to validate function signatures defined in agent prompts. A common failure is the verifier hallucinating non-existent constraints on function calls due to interference between model knowledge and examples in prompts. To address this, we devise a neural-symbolic verification system that prioritizes language models for validating solution completeness and pertinence while delegating fact checks to a symbolic system. We propose bootstrap-by-verification learning which combines massive agent trajectory sampling with our verification for self-training. Experiments on spreadsheet and web browsing benchmarks show the method's effectiveness.

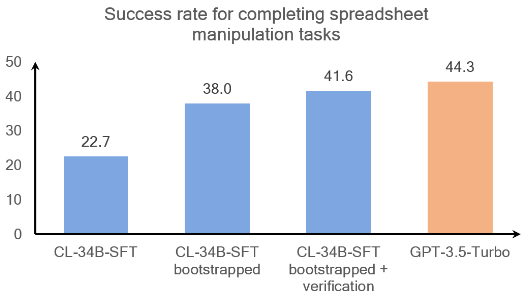


Figure 1: Our bootstrapping method verifies predictions of a base LLM, selects potentially correct ones, and uses the collected samples to fine-tune the LLM for specific tasks. Tested on the SheetCopilot (Li et al., 2023a) spreadsheet benchmark, the bootstrapped LLM shows significant improvement, approaching GPT-3.5-Turbo with the help of verifiers.

1 Introduction

Large language models (LLMs) have recently gained popularity as the engine for autonomous agents. Research explores leveraging LLMs beyond chatting, writing, and coding for general workplace applications. Proposed agent applications include problem solvers like Auto-GPT (Richards, 2023) and BabyAGI (Nakajima, 2023), gameplay agents like Voyager (Wang et al., 2023) and GITM (Zhu et al., 2023) and web surfing agents like WebGPT (Nakano et al., 2022) and Mind2Web (Deng et al., 2023).

Along with the prosperity of agent research, the need for LLM customization beyond prompting has surged and thus make the annotation for agent behavior of great importance. However, gathering expert trajectories for all possible tasks is impractical since modern language models are far more knowledgeable than any human generalist. For example, a spreadsheet agent could easily generate a formula like `VLOOKUP(C2, "A:B", 2, FALSE)`, which is used to search for the value in cell `C2` in the first column of the range `"A:B"` and returns a value in the same row from the second column. This complex formula requires serious efforts to come up with even for human experts. Therefore, modern LLMs are generally aligned via reinforcement learning via human feedback(RLHF) (Ouyang et al., 2022), a technique that could leverage the much weaker supervision signals produced by annotators ordering their preference of different LLM responses for the same request.

However, fine-tuning agents differs from chatbots in needing to handle a wide variety of tasks. While summarization and rewriting bots can leverage scalable annotation from individuals, it is much harder to imagine finding specialized annotators for niche skills like developing an EDA chip design copilot. The long tail of technical domains makes agent annotation far less scalable than for chatbots. Minimizing human effort in agent data collection is thus critical.

This work proposes a bootstrapping by verification learning method to generate annotation data for LLM agents without human effort. It is motivated by the insight that verifying a solution is often easier than generating one (RSA, 1978; Goldwasser et al., 1989; Cook, 1971). Though the key insight is proven, two main technical challenges arise in utilizing bootstrapping by verification learning for LLM agents. First, generating high-quality solution proposals using open-source models less capable than proprietary ones. We avoid distilling trajectories from strong proprietary models like GPT-4, to better assess method merits. Second, designing a procedure to reliably filter incorrect solutions and retain correct ones. Self-training requires high signal-to-noise ratio(SNR) for effective bootstrapping.

To address these challenges, we propose a two-step approach: massive agent trajectory[1] sampling (MATS) to generate diverse, high-quality solution proposals, and neural-symbolic verification (NSV) to filter out superfluous solutions. Specifically, our MATS obtains a large corpus of diverse proposed solutions by combining path eliminating rejection sampling and trajectory mutation sampling. We increase diversity through high temperature decoding, rejecting trajectories sharing prefix beyond a certain ratio with any existing trajectories. We also randomly mutate actions in existing trajectories and let the agent start generating right after the mutated actions. These methods ensure that our proposal solution pool is of high diversity. For increasing solution SNR, we employ both rule-based symbolic verifiers and LLM-based neural verifiers for rejecting superfluous solutions generated by massive sampling. Specifically we construct a set of verification functions that each is tasked for catching a specific type of error in the solution via symbolic pattern matching. Moreover, we prompt the neural verifiers for checking the completeness of the solution and if the precondiitons of certain actions are meet. Finally, we retrain our base model after the high quality solution corpus are obtain via standard instruction tuning approach.

To address these challenges, we propose a two-step approach: massive agent trajectory sampling (MATS) to generate diverse, high-quality solution proposals, and neural-symbolic verification (NSV) to filter out superfluous solutions. Specifically, our MATS obtains a large corpus of diverse proposed solutions by combining path eliminating rejection sampling and trajectory mutation sampling. We increase diversity through high temperature decoding, rejecting trajectories sharing prefix beyond a certain ratio with any existing trajectories. We also randomly mutate actions in existing trajectories and let the agent start generating right after the mutated actions. For NSV, we employ rule-based symbolic verifiers and LLM-based neural verifiers. Symbolic verifiers catch specific error types via pattern matching. Neural verifiers check completeness and action preconditions. After obtaining a high quality corpus, we retrain our base LLM with standard instruction tuning. This two-step MATS and NSV approach allows generating agent training data without human input.

Our contribution could be summarized as follows:

- We propose a new self-training method for LLM-based agents by leveraging the bootstrapping by verification approach.
- We devise a novel sampling method for sampling diverse agent trajectories for solution proposal and a neural-symbolic verification method for improving the signal-to-noise ratio of proposed solutions.
- We evaluate the proposed bootstrapping learning on multiple complex agent benchmarks with multi-step reasoning. Our method improves the codellama-34b based model by 15.3% on SheetCopilot (Li et al., 2023a) and by 6.9% on Mind2Web (Deng et al., 2023).

2 Related Works

2.1 LLM-based Agents

Benefiting from vast amounts of human text knowledge, large language models (LLMs) have exhibited a sign of human-level intelligence. Harnessing the impressive potential of LLMs, a new wave of research attempts to augment LLMs with external tools to build autonomous agents that are capable of solving complex tasks on behalf of humans. Notable examples include VisProg (Gupta and Kembhavi, 2022), HuggingGPT (Shen et al., 2023), ReAct (Yao et al., 2022), SheetCopilot (Li et al., 2023a), GITM (Zhu et al., 2023), Voyager (Wang et al., 2023), MetaGPT (Hong et al., 2023), and Coscientist (Boiko et al., 2023). These methods typically utilize in-context learning (Brown et al., 2020), allowing LLMs to flexibly acquire new skills and knowledge from relevant context and a few demonstrative examples in plain text, without additional training. Additionally, to enhance overall performance in various tasks that require reasoning and interaction, Chain of Thoughts (CoT) (Wei et al., 2022) is generally employed to elicit an interconnected flow of reasoning and decision-making from LLMs.

2.2 Self-Improving LLMs

As supervised fine-tuning and RLHF are both data-hungry, self-improving LLMs has recently

[1]We use solution and trajectory interchangeably thereafter.

gained attention as it is less reliant on human annotations. SPIN (Chen et al., 2024) iteratively boosts a weak model by reshaping the training process as a two-play game: the main player (the LLM after fine-tuning) seeks to differentiate the responses of the old LLM from human responses, while the opponent (the old LLM) generates responses as similar as possible to human ones. This method outperforms models trained with extra human data or AI feedback. Self-Rewarding (Yuan et al., 2024) shares a similar idea: LLMs act as instruction-following models generating responses and also evaluate their responses via LLM-as-a-Judge prompting, obtaining preference data used for fine-tuning. Instruction Backtranslation (Li et al., 2023b) similarly augments training data by generating and selecting synthetic instruction-output pairs using the target LLM. Different from these works, our method explores self-improving LLMs in interactive scenarios where an LLM is prompted as an agent that uses external tools to solve compositional tasks, such as manipulating computer software.

3 Bootstrapping via Verification

To bootstrap agent performance for LLMs, we assume access to a base chat or code language model, a set of seed task descriptions, and a symbolic engine for verifying the correctness of solutions.

As shown in Figure 2, we want the base model to both propose new tasks from seed tasks as well as generate a large number of verification solutions. We then use symbolic and model-based verifiers for grading all the solutions and find the most plausible ones for training the next iteration of agent models.

One full bootstrapping cycle includes a self-instructed task generation step, a solution generation step, a verification step, and a self-training step. We will elaborate each step in the following sections.

3.1 Self-Instructed Task Generation

To generate a large number of tasks and to avoid making design choices towards the test task set, we leverage the base model for proposing new task descriptions via Self-Instruct (Wang et al., 2022). As shown in Figure 2 (a), we randomly sample an environment setup from the task pool and prompt the base model to come up with new tasks that can be done in this environment according to few-shot seed task examples. To minimize potential data contamination, we use a distinct set of task environments(e.g., unseen websites for browsing) and perform task-level deduplication to remove tasks similar to existing ones. Other more advanced instruction generation approaches like Phi-1 (Gunasekar et al., 2023) and Magicoder (Wei et al., 2023) may also be applied but this is slightly out of the scope of this work.

3.2 Massive Agent Trajectory Sampling

To address the challenge of obtaining high-quality supervision signals without distilling proprietary models, we conduct massive solution sampling for our base models at a high temperature to sample high-quality solutions for task descriptions.

We argue that language models possess the ability to complete tasks while they probably underperform due to the lack of proper alignment. Although the potential performance gain underscored by best-of-n solution sampling for coding LLMs has been widely reported (Chen et al., 2021), how to materialize this potential without access to the oracle remains an open problem. Moreover, different from outputting a whole piece of code, agent models generally output a mix of chain-of-thought thinking steps and the actual action trajectories represented in function calls, how to effectively sampling diverse action trajectories remains an open problem.

We propose a path eliminating rejection sampling approach for leveraging the more structured action output for LLM-based agents compared with code generation. For a sampled trajectory with N actions $\mathcal{T}_N = \{A_1, A_2, \cdots, A_N\}$, we reject it if all prefix of it overlaps with existing solution beyond a certain threshold $\min_{t=0,\ldots,T} \mathcal{T}_t \cap \mathcal{T}_{\text{exist}} > \tau$.

Besides sampling based proposal generation, we also leverage trajectory mutation for increasing the solution diversity. For a random trajectory $\mathcal{T}_N = \{A_1, A_2, \cdots, A_N\}$ in the solution pool, we randomly replace one of its action A_n with an action A' uniformly sampled from the whole solution pool of the same task. We then use the prefix for LLM agents to continue sampling the remaining actions.

In Figure 4, we show that as the number of sampled solutions increases, the best-of-k success rate for agents completing tasks also improves steadily. Moreover, the potential of massive sampling holds for both a wide range of language models as well as heterogeneous agent tasks from different fields.

3.3 Neural-Symbolic Verifier for LLM Agents

After we validate the potential of our agents in terms of their ability to generate high-quality best-of-n samples, the next challenge is how to unleash their capability by converting the best-of-n performance into the best-of-1 performance, without dis-

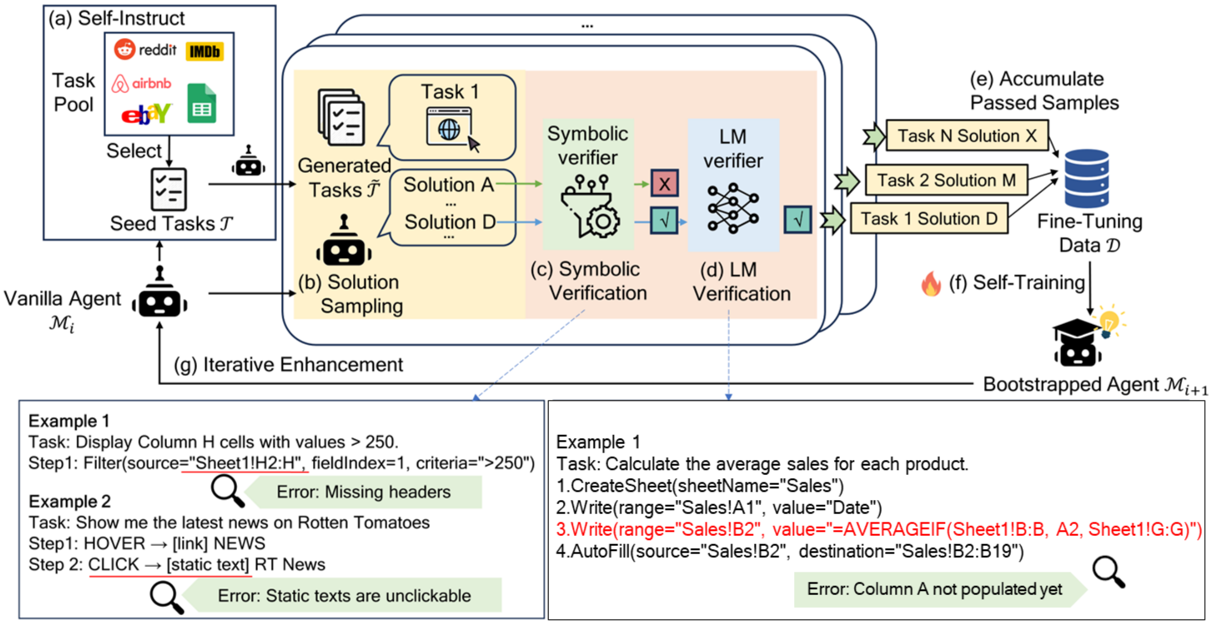


Figure 2: Overall framework of our proposed bootstrapping pipeline. The pipeline starts with an open-source LLM $\mathcal{M}_i$ and a small number of seed tasks $\mathcal{T}$. (a) Initially, $\mathcal{M}_i$ is prompted to generate more tasks using Self-Instruct (Wang et al., 2022), and then (b) $\mathcal{M}_i$ generates a number of candidate solutions to each task by interacting with the task-specific tools multiple times. (c) Subsequently, the candidate solutions undergo the symbolic verifier that employs verification functions to recognize potential errors in each step, which eliminates the solutions with easy-to-find errors. (d) Afterward, an LLM is prompted to verify the remaining solutions step-by-step to recognize hard-to-find errors probably related to the task semantics (Verification examples are shown at the bottom). (e) The task-solution pairs that pass the two verifiers are collected as the fine-tuning data $\mathcal{D}$, which is eventually used to (f) fine-tune $\mathcal{M}_i$ to obtain a more capable model $\mathcal{M}_{i+1}$. (g) This whole procedure will then be iterated resulting in a significantly improved LLM agent.

tillation or large-scale human labeling.

A neural-symbolic verifier is used for evaluating the correctness of model-generated solutions. In addition to utilize language models for verification like existing generate & rank works for math word problems (Shen et al., 2021; Cobbe et al., 2021), we further harness the power of symbolic verification in this work.

We combine the rigor of a symbolic system and the real-world understanding of language models for verification. As shown in Figure 7, agents make different types of errors. Errors like calling wrong APIs or referencing null objects are easily verifiable by the symbolic engine while errors like choosing unrelated APIs or hallucinating meaningless actions are more suitable for language model verifiers.

Symbolic Verification We borrow the idea of symbolic verification from traditional software analysis and formal verification systems (King, 1976; Clarke et al., 1986). Instead of building a full-fledged symbolic execution engine that could validate preconditions, post conditions, and invariants for a piece of code, our symbolic verifier simply consists an action parser and a set of argument verification functions. The action parser breaks up action function calls into arguments and a set of verification functions validates each argument both syntactically and semantically. Each symbolic verification function is tailored based on explicit logic for a specific task (e.g. spreadsheet and browser) and the syntax of the corresponding output APIs. For instance, in the context of the spreadsheet task, a verification function is designed to check the data integrity for a source range (the cells start from row 2 in column H in Sheet1) of an `<Filter(source='Sheet1!H2:H',fieldindex=1,criteria='>250')>` action. Similarly, for the browsing task, a verification function is tasked with checking whether the target element `[static text] RT News` of a `<CLICK on [static text] RT News>` action clickable. The bottom left corner of Figure 2 visually depicts these two functions.

The development of these verification functions begins with running the agent on the validation dataset, where we then analyze and catalog errors from the agent's running log. This hands-on analysis allows us to discern patterns and commonalities of errors, informing the creation of a symbolic engine.

Symbolic-aided Neural Verification Different from existing works that solely rely on language models for verification via prompting (Yuan et al., 2024) and fine-tuning (Cobbe et al., 2021), we propose a symbolic-aided neural verification approach. Specifically, we first check the initial solution via the verification functions. We then incorporate into prompt the symbolic checking results to conduct further checking by language verifier for solutions passed by symbolic checking. By hinting language models with symbolic checking results, we can avoid the language model hallucinating wrong replies about facts already been verified. Besides, the language models can now focus on generating verification that has not been verified symbolically, effectively reducing the solution space. Our language model verifier only checks the completeness of solutions and whether the action choice is aligned with the task instruction. By delegating only high tasks with requirements of real-world understanding to the LLM verifier, we can greatly simplify the task requirement and reduce the risk of hallucination.

3.4 Verification-based Self-Training

The final step in our bootstrapping cycle is to utilize verified solutions to further fine-tune the base model. This verification-based self-training learns from past successful cases, benefiting from the self-capabilities unleashed by a symbolic-aided neural verifier on massive solution sampling. We initiate the process by conducting massive sampling on the base model, collecting solutions that have passed through the Neural-Symbolic Verifier to constitute our training set. We then fine-tune the base model on this dataset. In the experiment, we prove that these verified solution play a key role in bootstrapping base model.

4 Experiments

4.1 Experimental Settings

Base Model. We adopt two open-source models, CodeLlama-34B-SFT (Rozière et al., 2023)[2] and Magicoder-S-DS-6.7B (Wei et al., 2023)[3].

Task Dataset We use two challenging public benchmarks, SheetCopilot (Li et al., 2023a) and Mind2Web (Deng et al., 2023): 1) The SheetCopilot dataset contains 28 evaluation workbooks and 989 spreadsheet manipulation tasks, categorized into 768 training samples and 221 test samples, that are applied to these workbooks. Each task specifies a high-level request, involving standard spreadsheet operations.

2) The Mind2Web dataset consists of web-browsing tasks derived from 137 websites across various domains. It assesses the ability of agents to follow human instructions for completing complex tasks in web environments. Each step of a task is evaluated independently with the ground truth action history provided, prompting an agent to predict either Click [Id], Type [Id] [Value], or Select [Option]. The cross-website split of this dataset is used in our experiments. Examples of the two benchmarks are shown in Figure 3.

Evaluation Metrics SheetCopilot benchmark uses Exec@1 to measure the percentage of solutions executed without throwing exceptions and Pass@1 to evaluate functional correctness (Chen et al., 2021). To fully evaluate the potentials of the LLM agents, we extend these two metrics to Exec@k and Pass@k. The former is defined as the probability that at least one of the top k-generated solutions for a single task can be executed without exceptions. The latter is similarly defined. As each step is evaluated independently, we use Element Accuracy (Elem. Acc.) and Step Success Rate (Step SR) in the Mind2Web benchmark. The former calculates the proportion of the predicted elements that match ground truths and the latter calculates the proportion of predicted steps whose predicted element and operation are both correct. Likewise, we also extend these metrics to Elem. Acc.@k and Step SR@k by generating multiple predictions for each test sample. All Pass@1 values are calculated at temperature=0.0 while all Pass@k at temperature=1.0.

Compared Methods On the SheetCopilot benchmark, we compare the performances of the SheetCopilot agent with CodeLlama-34B-SFT bootstrapped with our method and GPT-3.5-Turbo which is originally used. On the Mind2Web benchmark, we compare the performances of the agent provided in this benchmark with verifier-equipped CodeLlama-34B-SFT and GPT-3.5-Turbo/GPT-4 as its backend. We also compare with Synapse (Zheng et al., 2023) which uses few-shot in-context exemplars semantically similar to the task at hand to prompt GPT-3.5-Turbo to generate the next action.

Self-Training Details We test the proposed bootstrapping via verification on the SheetCopilot dataset. At each iteration, a target LLM is run on the training split of the SheetCopilot benchmark 6 times, generating 4608 task-solution pairs, each

[2]https://huggingface.co/Phind/Phind-CodeLlama-34B-v2
[3]https://huggingface.co/ise-uiuc/Magicoder-S-DS-6.7B

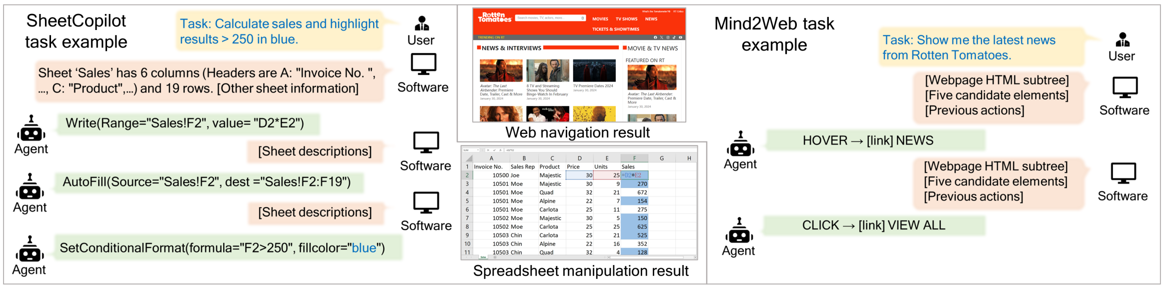


Figure 3: Examples of the used benchmarks. Left: a SheetCopilot task that requires calculating product sales and highlighting key data using conditional formatting. Right: a Mind2Web task that requires finding news on Rotten Tomatoes, a review-aggregation website for movies and television.

of which is a dialog between the user/software and the agent (An example is shown in the Appendix). Filtered by the two proposed verifiers, each of the passed pairs is decomposed into multiple training samples. Specifically, a training sample is a discrete step in the solution process (i.e., a turn in the dialog), inclusive of its history. We fine-tune the full parameters of the target LLM through supervised fine-tuning, using the collected training samples. An iteration of bootstrapping is run for three epochs. The loss is only computed on target tokens instead of complete sequences. The learning rate is $1e-5$ and the batch size is 1. More details are included in the appendix.

4.2 Comparing with Existing Works

Spreadsheet Manipulation We test our method using CodeLlama-34B-SFT as the target model to be bootstrapped and compare it with the raw model as well as a baseline (Li et al., 2023a) using GPT-3.5-Turbo. Note that these compared models are used as the SheetCopilot agent backend to run evaluation. The results in Table 1 show that the Pass@50 of CodeLlama-34B-SFT is notably higher than the Pass@1 of GPT-3.5-Turbo, indicating that this open-source model is capable of sampling functionally correct solutions. After undergoing 1 iteration of bootstrapping, CodeLlama-34B-SFT-Iter1 achieves significant performance gains, outperforming its raw model by 15.3 Pass@1. Using the proposed verifiers to aid the bootstrapped model, the pass@1 of this model is even close to that of GPT-3.5-Turbo. These results suggest that the target model is progressively aligned with the desirable behavior required by spreadsheet manipulation tasks through fine-tuning with the training samples filtered by the proposed verifiers.

Web Browsing For the Mind2Web benchmark, we compare the performances of the target model, CodeLlama-34B-SFT, and GPT-3.5-Turbo/GPT-4 by using these models as the backend of MindAct, the agent provided in this benchmark. We report the performances of the target model that utilizes the proposed verifiers to find the functionally correct solution out of 20 sampled predictions. We also compare with Synapse (Zheng et al., 2023) which uses few-shot in-context exemplars semantically similar to the task at hand to prompt GPT-3.5-Turbo to generate the next action. The results in Table 2 illustrate that the target model, CodeLlama-34B-SFT, enjoys clear improvement when generating 20 predictions (best of 20) for each test sample although it shows weak web-browsing capability when generating only one prediction. This best-of-20 model also outperforms MindAct with GPT-3.5-Turbo and GPT-4. After being equipped with the proposed verifiers, the target model (CL + Verif.) achieves performance higher than that of the target model, outperforming MindAct (GPT-3.5) and close to Synapse (GPT-3.5). In summary, the results on the two benchmarks indicate that open-source LLMs possess the potential of being lifted to the level of proprietary LLMs on specific domains and that the proposed bootstrapping with verification is capable of unleashing this potential.

4.3 Evaluating Best-of-K Sampling

LLMs To assess the generalizability of our method, we test diverse open-source LLMs by plotting curves that illustrate the metrics calculated by sampling multiple predictions at different k. Apart from CodeLlama-34B-SFT, we test a smaller coding model, Magicoder-6.7B, and a small chatting model, Llama2-7B-chat, to observe the potential of various target models. Figure 4(a) demonstrates that CodeLlama-34B-SFT significantly outperforms GPT-3.5-Turbo when $k > 6$. Additionally, despite an extremely low Pass@1, the smaller Magicoder-6.7B demonstrates Pass@50 comparable to Pass@1 of GPT-3.5-Turbo, which indicates

Table 1: Overall performance on the SheetCopilot benchmark. This table compares the target model bootstrapped with our proposed method and three proprietary LLMs. When using the verifiers, we keep sampling predicted solutions until one solution passes verificaiton or we exceed the sampling limit (50 solutions), and then consider the tasks with solutions found within the limit as successful. The target model, CodeLlama-34B-SFT, achieves impressive Exec@50 and Pass@50 which substantially surpass Exec@1 and Pass@1 of the three proprietary LLMs. When aided by the verifiers, the target model obtains higher Pass@1. Additionally, our bootstrapping method unleashes the model's potential using verification-aided self-training, lifting this target model to a higher level. Using the verifiers to augment the bootstrapped model introduces further improvement in Pass@1. 10% means that the experiments are conducted with 10% of the test samples due to the formidable cost of the LLM APIs.

Model	Exec@1	Pass@1	Exec@50	Pass@50
CodeLlama-34B-SFT	94.1	22.7	100.0	60.6
CodeLlama-34B-SFT w/ verifiers	94.1	34.5	-	-
CodeLlama-34B-SFT-iter1	96.4	38.0	100.0	64.3
CodeLlama-34B-SFT-iter1 w/ verifiers	85.1	41.6	-	-
GPT-3.5-Turbo (Li et al., 2023a)	87.3	44.3	-	-
GPT-4 (10%) (Li et al., 2023a)	65.0	55.0	-	-
Claude (10%) (Li et al., 2023a)	80.0	40.0	-	-

Table 2: Overall performance on the cross-website split of the Mind2web benchmark. The target model, CodeLlama-34B-SFT (CL), is weaker than all of the three compared methods. However, this model obtains notably high metrics when sampling 20 predictions (best of 20). Using the verifiers (Verif.), the target model also achieves performance gain, outperforming MindAct (GPT-3.5).

Model	Elem. Acc.	Step SR
MindAct (CL)	14.7	12.1
MindAct (CL + Best of 20)	**54.4**	**34.8**
MindAct (CL + Verif.)	22.6	19.0
MindAct (GPT-3.5)	19.3	16.2
MindAct (GPT-4)	35.8	30.1
Synapse (GPT-3.5)	28.8	23.4

that this smaller model is also likely to be lifted to a GPT-3.5-Turbo level by leveraging our bootstrapping method. This trend also appears on the Mind2Web benchmark. CodeLlama-34B-SFT outperforms GPT-4 when $k > 5$ while the other two smaller models achieve performances comparable to, or even higher than the level of GPT-3.5-Turbo.

Overall, the above results on diverse metrics and benchmarks suggest that it is possible to leverage our bootstrapping method to elevate open-source LLMs to a similar level of proprietary LLMs.

4.4 Ablation Studies

4.4.1 Evaluating Verifiers

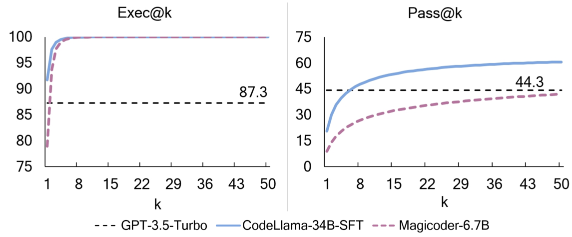
We justify the efficacy of the proposed verification process by 1) inspecting its precision and recall, and 2) equipping open-source LLMs with the verifiers when tested on the two benchmarks.

Firstly, we apply the proposed verifiers to the results of CodeLlama-34B-SFT in 4.2, and calculate the precision and recall using the verification results. The precisions for the functionally correct samples and the failed ones are 0.40 and 0.86, respectively. The recalls for the functionally correct samples and the failed ones are 0.53 and 0.79, respectively. We can see that the verifiers achieve higher precision and recall of recognizing failed solutions despite the lower values for the successful ones. As the verifiers are designed to recognize potential errors in generated solutions and to reject as many potentially failed solutions as possible, instead of picking correct ones, this imbalance phenomenon can be expected. We notice that the precision is higher than the recall for recognizing failed solutions, which is because our verifiers are designed to be general enough to recognize common errors. As error types are difficult to enumerate, it is almost impossible to invent all possible rules used to recognize all error types. Therefore, the verifiers can find erroneous solutions precisely while likely to miss the ones with elusive errors. Symmetrically, the recall is higher than the precision for finding successful solutions since another goal of our verifiers is to recognize as many errors as possible without missing successful solutions. Therefore, our verifiers may mistakenly judge failed solutions as correct so as to not miss potentially successful ones.

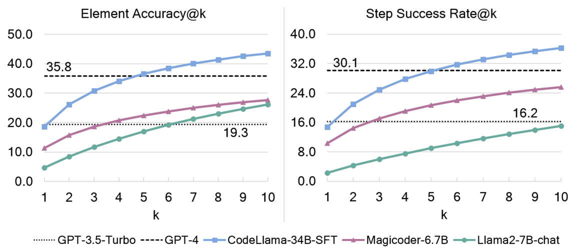
To assess the efficacy of the verifiers, we evaluate the target model with and without the proposed verifiers. For the model without verification, the temperature is 0.0; for the model with verification, the temperature is 1.0 and we sample predictions until one prediction passes the verification. Table 3 shows that the target model, CodeLlama-34B-SFT, obtains higher Pass@1 when equipped with the

Table 3: Ablation studies on the proposed verifiers. CodeLlama-34B-SFT, Magicoder-6.7B, and Llama2-7B-chat are used as the target LLMs. When evaluated without the proposed verifiers, the inference temperature is set to 0.0. When verifiers are used, the temperature is set to 1.0.

Symbolic Verifier	LM Verifier	SheetCopilot Exec@1	SheetCopilot Pass@1	Mind2Web Elem. Acc.@1	Mind2Web Step SR@1
		94.1	22.7	14.8	12.1
✓		97.3	33.1	19.2	16.3
✓	✓	91.4	34.5	22.6	19.0



(a) Evaluation on the SheetCopilot benchmark.



(b) Evaluation on the Mind2Web benchmark.

Figure 4: Experiments of Best-of-K sampling. We test different open-source LLMs on the two benchmarks by calculating the metrics via best-of-k sampling. On the SheetCopilot benchmark, the largest model, CodeLlama-34B-SFT surpasses GPT-3.5-Turbo when $k > 6$ while the smaller Magicoder-6.7B becomes comparable to GPT-3.5-Turbo when $k = 50$. The Mind2Web benchmark also exhibits similar trends: the three open-source LLMs obtain consistent improvements when k increases, with CodeLlama-34B-SFT outperforming GPT-4 and the other obtaining performances comparable to, and even surpassing, that of GPT-3.5-Turbo.

symbolic verifier. Using both verifiers leads to a slightly higher Pass@1. Adding the LM verifier reduces Exec@1, possibly because this verifier is strict, rejecting several potentially correct predictions. On the Mind2Web benchmark, the proposed verifiers also bring consistent improvements. These results show that the proposed verifiers are beneficial for improving open-source LLMs prompted as autonomous agents.

4.4.2 Evaluating Self-Training

To see to what extent we can enhance the ability of open-source LLMs in specific domains, we bootstrap the target model, CodeLlama-34B-SFT, and

Table 4: The impact of solution SNR on self-training performance.

Model	Exec@1	Pass@1
CodeLlama34B-SFT	94.1	22.7
Self-Instruct w/ Verifiers (2185)	**96.4**	**38.0**
Self-Instruct w/o Verifiers (2185)	92.8	21.7

observe the variation in its performance. The result in Figure 5 demonstrates that the target model obtains a substantial performance gain with one iteration of self-training, achieving a Pass@1 near the level of GPT-3.5-Turbo. This result validates that our bootstrapping method can effectively generate high SNR solutions to improve the model.

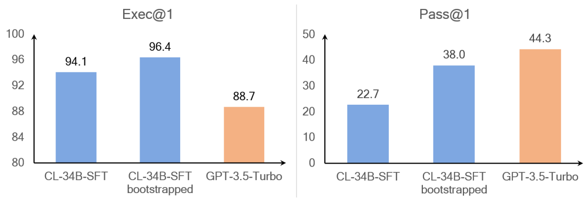


Figure 5: The performance of CodeLlama-34B-SFT on the SheetCopilot benchmark after bootstrapping. With one iteration of bootstrapping, the target model (CL-34B-SFT) witnesses significant improvement in both metrics, increasing Pass@1 by 15.3, near the level of GPT-3.5-Turbo.

5 Conclusion

We present bootstrapping by verification learning for LLM-based agents in this work. Our approach combine a new massive agent trajectory sampling method and a neural-symbolic verification approach for generating high signal-to-noise solutions for self-training our base model. Experiments on multi-steps spreadsheet manipulation and web surfing tasks demonstrate the effectiveness of the proposed methods. We hope this work could bring more research interests into studying how to align agent behavior without large-scale human annotation.

6 Limitations

Our method is evaluated on only multi-step benchmarks of agent tasks. The importance and significance for automatic alignment for those one-step benchmarks like ToolLlama (Qin et al., 2023) is not studied in this work.

References

1978. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126.

Daniil A. Boiko, Robert MacKnight, Ben Kline, and Gabe Gomes. 2023. Autonomous chemical research with large language models. *Nature*, 624:570 – 578.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901.

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

Zixiang Chen, Yihe Deng, Huizhuo Yuan, Kaixuan Ji, and Quanquan Gu. 2024. Self-play fine-tuning converts weak language models to strong language models. *arXiv preprint arXiv:2401.01335*.

Edmund M Clarke, E Allen Emerson, and A Prasad Sistla. 1986. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems.

Stephen A. Cook. 1971. The complexity of theorem-proving procedures. *Proceedings of the third annual ACM symposium on Theory of computing*.

Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *NeurIPS*, 35:16344–16359.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. 2023. Mind2web: Towards a generalist agent for the web. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Shafi Goldwasser, Silvio Micali, and Charles Rackoff. 1989. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. 2023. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*.

Tanmay Gupta and Aniruddha Kembhavi. 2022. Visual programming: Compositional visual reasoning without training. *CVPR*, pages 14953–14962.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2023. Metagpt: Meta programming for a multi-agent collaborative framework.

James C King. 1976. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394.

Hongxin Li, Jingran Su, Yuntao Chen, Qing Li, and Zhaoxiang Zhang. 2023a. Sheetcopilot: Bringing software productivity to the next level through large language models. In *NIPS*.

Xian Li, Ping Yu, Chunting Zhou, Timo Schick, Luke Zettlemoyer, Omer Levy, Jason Weston, and Mike Lewis. 2023b. Self-alignment with instruction backtranslation. *arXiv preprint arXiv:2308.06259*.

Yohei Nakajima. 2023. Babyagi. `https://github.com/yoheinakajima/babyagi`. GitHub repository.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. 2022. Webgpt: Browser-assisted question-answering with human feedback.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Gray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis.

Toran Bruce Richards. 2023. Auto-gpt. https://github.com/Significant-Gravitas/Auto-GPT. GitHub repository.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code llama: Open foundation models for code.

Jianhao Shen, Yichun Yin, Lin Li, Lifeng Shang, Xin Jiang, Ming Zhang, and Qun Liu. 2021. Generate & rank: A multi-task framework for math word problems. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 2269–2279, Punta Cana, Dominican Republic. ACL.

Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. 2023. Hugging-GPT: Solving AI tasks with chatGPT and its friends in hugging face. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2022. Self-instruct: Aligning language model with self generated instructions. *arXiv preprint arXiv:2212.10560*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, brian ichter, Fei Xia, Ed H. Chi, Quoc V Le, and Denny Zhou. 2022. Chain of thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. 2023. Magicoder: Source code is all you need.

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

Weizhe Yuan, Richard Yuanzhe Pang, Kyunghyun Cho, Sainbayar Sukhbaatar, Jing Xu, and Jason Weston. 2024. Self-rewarding language models. *arXiv preprint arXiv:2401.10020*.

Longtao Zheng, Rundong Wang, Xinrun Wang, and Bo An. 2023. Synapse: Trajectory-as-exemplar prompting with memory for computer control. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*.

Xizhou Zhu, Yuntao Chen, Hao Tian, Chenxin Tao, Weijie Su, Chenyu Yang, Gao Huang, Bin Li, Lewei Lu, Xiaogang Wang, et al. 2023. Ghost in the minecraft: Generally capable agents for open-world enviroments via large language models with text-based knowledge and memory. *arXiv preprint arXiv:2305.17144*.

A Appendix

A.1 More Training details

Flash-attention (Dao et al., 2022) and bfloat16[4] are also utilized to accelerate training.

A.2 Task Solution Example

We show a example of the SheetCopilot benchmark used in our experiments in Figure 6. The left column of the figure shows that the agent generates a step-by-step solution according to the sheet state feedback and correctly revises its mistakes using the external atomic action document as well as the error feedback. The incorrect arguments are marked with red rectangles. The right column shows that the sheet state changes corresponding to each step on the left.

A.3 Error Types Recognized in Verification

To fully assess the effect of the proposed verifiers, we display the proportions of the error types recognized in the verification in a Sankey diagram (Figure 7). A large percentage of the errors are found by the symbolic verifier, which include Referring invalid objects, Incomplete Data, Meaningless Actions, Argument errors, and Other common-sense errors. A small number of errors are recognized by the LLM verifier since these errors related to the task semantics occur less frequently.

[4]Bfloat16 is a floating-point number format with 16 bits, striking a balance between the range of traditional 32-bit floating-point numbers and the memory efficiency of 16-bit floating-point numbers.

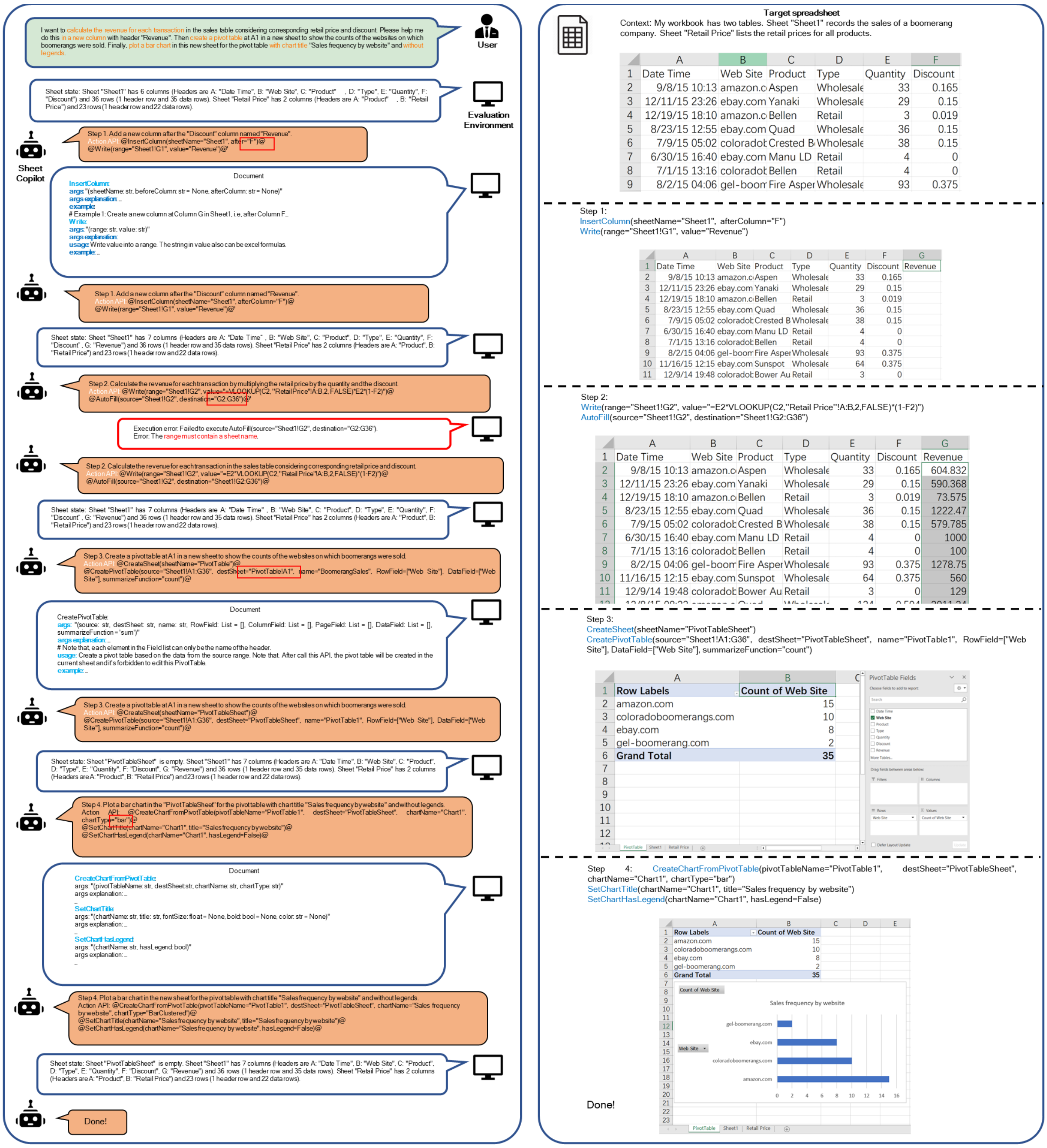


Figure 6: A task solution example of the SheetCopilot benchmark.

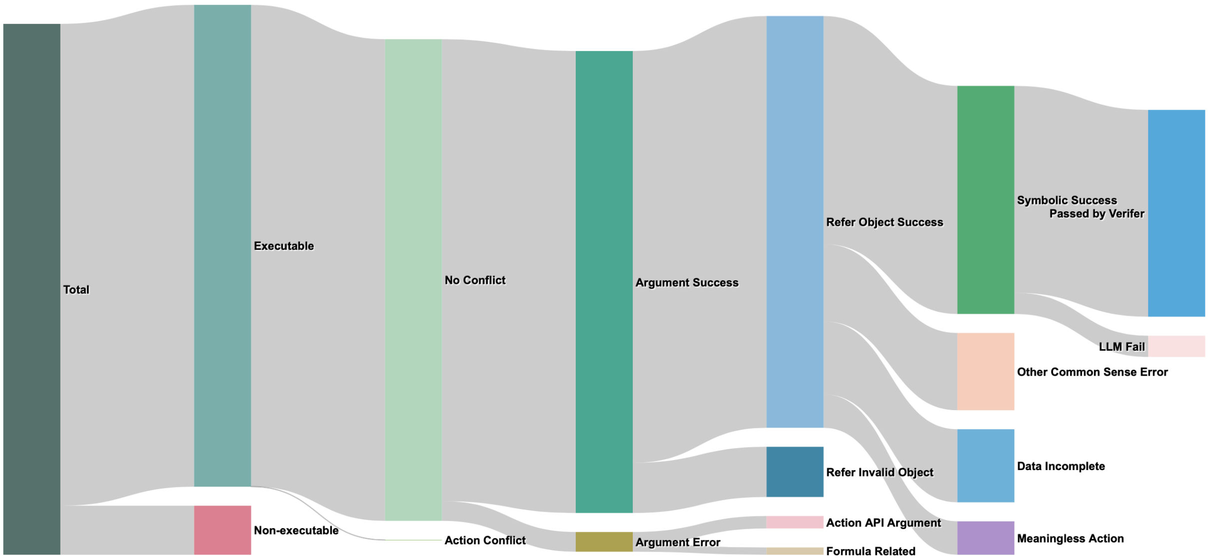


Figure 7: Error breakdown for the codellama-34b-sft model on SheetCopilot tasks.