Self-Generated In-Context Examples Improve LLM Agents for Sequential Decision-Making Tasks

Vishnu Sarukkai [1] **Zhiqiang Xie** [1] **Kayvon Fatahalian** [1]

Abstract

Large Language Model (LLM) agents typically require extensive task-specific knowledge engineering. Instead, we investigate a self-improvement approach where agents learn from their own successful experiences without human intervention. Our method builds and refines a database of self-generated trajectories for in-context learning. Even naive accumulation of successful trajectories yields substantial performance gains across ALFWorld (73% to 89%), Wordcraft (55% to 64%), and InterCode-SQL (75% to 79%). We further enhance performance with 1) database-level curation using population-based training, and 2) exemplar-level curation that selectively retains trajectories based on their utility. On ALFWorld, our method achieves 93% success—surpassing approaches using more powerful LLMs. Our approach demonstrates that agents can autonomously improve through self-collected experience, without labor-intensive knowledge engineering.

1. Introduction

When creating LLM agents for sequential decision-making tasks, practitioners often improve performance through task-specific knowledge engineering—requiring substantial human effort through prompt tuning (Wei et al., 2022), handcrafted examples (Brown et al., 2020; Wei et al., 2023), or custom action spaces (Chen et al., 2024; Yang et al., 2024).

We investigate an alternative: enabling LLM agents to autonomously bootstrap performance by leveraging their own successful experiences via in-context learning. We focus on how ReAct-style agents (Yao et al., 2023) can construct and refine a database of self-generated examples (Kagaya et al., 2024; Zhou et al., 2024), addressing two key challenges: collecting high-quality trajectories and strategically curating the most valuable ones for future retrieval.

Even naive database accumulation improves test-set performance from 73% to 89% on ALFWorld, 55% to 64% on Wordcraft, and 75% to 79% on InterCode-SQL. We further propose two enhancements: (1) database-level curation that propagates high-performing example collections, and (2) exemplar-level curation that identifies helpful trajectories based on their empirical utility (Akyürek et al., 2022; Von Oswald et al., 2023; Agarwal et al., 2024). These approaches improve success rates on ALFWorld to 93%—surpassing approaches that use more powerful LLMs and handcrafted components (Chen et al., 2024), as well as hierarchical approaches (Fu et al., 2024). The improvement exceeds the boost from upgrading from gpt-4o-mini to gpt-4o, highlighting self-generated in-context examples as a dimension for scaling test-time compute.

2. Preliminaries

Sequential Decision-Making Tasks We focus on multi-step sequential decision-making tasks where agents produce a series of actions based on environmental observations. We assume a standard POMDP setup where an agent, given a task goal g, interacts with environment $\mathcal{E}$ for up to T timesteps. At each timestep t, the agent receives observation o_t, takes action a_t, and $\mathcal{E}$ transitions to the next state. We consider sparse-reward environments where success is only determined at episode end, a standard setting in prior work (Yao et al., 2023; Zhao et al., 2024; Fu et al., 2024; Chen et al., 2024).

ReAct-style Agent Loop Our work assumes a ReAct-style (Yao et al., 2023) agent architecture with recent best practices for in-context retrieval (Kagaya et al., 2024; Zhou et al., 2024). As shown in Alg. 1, two key components differentiate our implementation from basic ReAct: (1) an initial planning step before execution begins (line 3), shown to boost performance (Kagaya et al., 2024; Song et al., 2023; Zhao et al., 2024; Chen et al., 2024), and (2) dynamic retrieval of different trajectory segments for each decision point (Kagaya et al., 2024; Zhou et al., 2024).

[1]Stanford University. Correspondence to: Vishnu Sarukkai <sarukkai@stanford.edu>.

Proceedings of the 42^{nd} International Conference on Machine Learning, Vancouver, Canada. PMLR 267, 2025.

Algorithm 1 ReAct-style Agent Loop

function AGENT($g, \mathcal{D}, \mathcal{E}, T$)
  $C_p \leftarrow$ RETRIEVE($\mathcal{D}$, keys $= [g]$)
  $p \leftarrow \text{LLM}_{\text{plan}}(g, C_p)$ {Generate initial plan}
  Initialize $\tau \leftarrow (g, p, \{\}, -)$; $o_1 \leftarrow \mathcal{E}.\text{OBS}()$
  $C_1 \leftarrow$ RETRIEVE($\mathcal{D}$, keys $= [g, p, o_1]$)
for $t = 1$ to T **do**
  $r_t \leftarrow \text{LLM}_{\text{reason}}(\tau, o_t, C_t)$ {Generate reasoning}
  $C_{t+1} \leftarrow$ RETRIEVE($\mathcal{D}$, keys $= [g, p, r_t]$)
  $a_t \leftarrow \text{LLM}_{\text{act}}(\tau, o_t, r_t, C_{t+1})$ {Decide action}
  o_{t+1}, done, $s \leftarrow \mathcal{E}.\text{STEP}(a_t)$ {Execute action}
  $\tau \leftarrow \tau \cup (o_t, r_t, a_t)$
  if done **then**
    return $(g, p, \{(o_i, r_i, a_i)\}_{i=1}^{t}, s)$
return $(g, p, \{(o_i, r_i, a_i)\}_{i=1}^{T}, 0)$ {Timeout}

The agent operates through three LLM-based functions: $\texttt{LLM}_{\text{plan}}$ generates a high-level plan p, $\texttt{LLM}_{\text{reason}}$ processes observations to produce reasoning, and $\texttt{LLM}_{\text{act}}$ determines actions. Our contribution focuses specifically on constructing and refining the trajectory database powering this retrieval mechanism, without relying on task-specific prompting or custom observation/action spaces (Zhou et al., 2024; Yang et al., 2024; Chen et al., 2024).

3. Problem Statement

Given the ReAct-style agent described in Sec. 2, our goal is to construct a trajectory database that maximizes agent performance on sequential decision-making tasks. We focus on building and refining the example database accessed by the agent's retrieval mechanism.

Formally, we aim to construct a trajectory database $\mathcal{D}$ where each trajectory $\tau \in \mathcal{D}$ captures a complete task attempt: $\tau = (g, p, \{(o_t, r_t, a_t)\}_{t=1}^{T}, s)$. We aim to maximize agent performance across tasks $\mathcal{T}$: $\mathcal{D}^* = \arg\max_{\mathcal{D}} \mathbb{E}_{g \sim \mathcal{T}}[\texttt{Success}(\texttt{Agent}(g, \mathcal{D}, \mathcal{E}, T))]$, where $\texttt{Success}()$ returns the binary outcome s. We assume that we are given: (1) $\mathcal{D}$ initialized with a small number of human-generated trajectories, (2) a descriptor of the action space, and (3) access to training tasks drawn from $\mathcal{T}$ that the agent can attempt—all typical assumptions for ReAct-based agents (Yao et al., 2023; Kagaya et al., 2024; Zhao et al., 2024; Fu et al., 2024; Chen et al., 2024).

4. Related Work

In-context learning for agent improvement Despite the popularity of reinforcement learning approaches for improving agent capabilities (Bai et al., 2022; Rafailov et al., 2023; Jaech et al., 2024; Guo et al., 2025), in-context learning offers distinct advantages: model-agnostic portability across LLMs, efficiency in low-sample regimes (Wei et al., 2023; Bertsch et al., 2024), and accessibility when weight modification is infeasible. Both empirical and theoretical work shows that in-context performance can scale with additional examples (Akyürek et al., 2022; Von Oswald et al., 2023; Bertsch et al., 2024; Agarwal et al., 2024). We focus on maximizing limited examples through in-context methods, while hypothesizing that database quality critically influences performance scaling.

In-context self-improvement of LLM Agents Self-improvement methods for LLM agents either aim to solve one task (performing search/optimization) or transfer knowledge from prior tasks to novel ones (generalization). Single-task approaches scale the number of sampled solutions(Brown et al., 2024; Wang et al., 2024a;b) or incorporate feedback from failed attempts(Shinn et al., 2023). Knowledge transfer approaches include abstraction-based methods like ExpeL(Zhao et al., 2024) and AutoGuide(Fu et al., 2024), while others employ task-specific information—RAP(Kagaya et al., 2024) uses task-specific prompts and AutoManual(Chen et al., 2024) constructs task-specific state and action spaces. Rather than developing complex architectures or leveraging task-specific information, we focus on identifying which trajectories most contribute to successful outcomes as in-context examples.

5. Methods

We now discuss three algorithms for constructing database $\mathcal{D}$ using a continual collection approach.

5.1. *Traj-BS*: Constructing a Database of Previously-Solved Tasks

Our trajectory-bootstrapping algorithm Traj-BS constructs a trajectory database $\mathcal{D}$ by collecting successful agent experiences. We start with a minimal set of human-provided exemplars, then grow the database as the agent successfully completes training tasks. This process creates a positive feedback loop where successful examples help the agent solve new tasks, generating more successful examples.

Traj-BS operates on principles similar to reward-weighted regression in reinforcement learning (Peters & Schaal, 2007), where only successful trajectories ($s = 1$) are stored. This filtering ensures the agent learns from positive examples while avoiding potentially misleading failed attempts.

5.2. *+DB-Cur*: Database-Level Data Curation

Traj-BS exhibits unpredictable performance variation across training trials, even when following identical collection procedures (App. E, Fig. 2). This variance arises from: (1) the stochasticity of LLM outputs creating different initial trajectories, and (2) an amplification effect where early differ-

Algorithm 2 Database Curation Logic for +DB-Cur

procedure +DB-CUR($\{\mathcal{D}_1, \mathcal{D}_2, ..., \mathcal{D}_N\}$, $interval$)
  Initialize performance metrics $\{m_1, m_2, ..., m_N\}$ for each database
  for $t = 1$ to T_{train} **do**
    for each $i = 1$ to N **in parallel do**
      Execute task t using database $\mathcal{D}_i$
      if task successful **then** add trajectory to $\mathcal{D}_i$
      Update rolling performance metric m_i on recent tasks
    if $t = 10 \times 2^j$ for any $j \in \mathbb{N}$ **then**
      Sort databases by rolling performance on recent tasks
      Replace worst database with copies of best

Algorithm 3 Database Construction from Top Exemplars for +EX-Cur

procedure +EX-CUR($\{\mathcal{D}_1, \mathcal{D}_2, ..., \mathcal{D}_N\}$, T_{train})
  $\mathcal{D}_{\text{composite}} \leftarrow \emptyset$
  Compute quality metric $Q(\tau)$ for each trajectory $\tau \in \bigcup_{i=1}^{N} \mathcal{D}_i$
  for each task $t \in T_{\text{train}}$ **do**
    $T_t \leftarrow$ {successful trajectories for task t across all databases}
    if T_t not empty **then**
      Select top-1 trajectory from T_t by quality metric Q
      Add selected trajectory to $\mathcal{D}_{\text{composite}}$
  return $\mathcal{D}_{\text{composite}}$

ences lead to wide performance variation. This observation motivates a data curation strategy inspired by population-based training (Jaderberg et al., 2017). We introduce *+DB-Cur*, a population-based training algorithm (Alg. 2) to identify and propagate effective databases during bootstrapping.

+DB-Cur maintains N database instances that accumulate successful trajectories independently. At size thresholds, it evaluates performance based on recent success rates and replaces the worst-performing database with a copy of the top one. The key insight is that database quality emerges from collective properties—like coverage and complementarity across examples—not just individual trajectory quality.

5.3. *+EX-Cur*: Exemplar-Level Data Curation

While database-level curation identifies complementary trajectory sets, discarding whole databases can eliminate valuable trajectories. We introduce *+EX-Cur*: identifying high-quality exemplars across database instances based on their empirical utility. We define a retrieval-weighted quality metric:

$$Q(\tau) = \frac{\sum_{i \in \mathcal{R}(\tau)} o_i \cdot f_i(\tau)}{\sum_{i \in \mathcal{R}(\tau)} f_i(\tau)} \tag{1}$$

where $\mathcal{R}(\tau)$ is the set of tasks for which trajectory τ was retrieved, o_i is the binary outcome of task i, and $f_i(\tau)$ is the retrieval frequency during task i.

Alg. 3 outlines the process: for each training task, it identifies all successful trajectories across database instances and selects the exemplar with highest Q value.

6. Experiments

We evaluate our database construction methods through experiments addressing three key questions: 1) How does task success rate scale with increasing database size? 2) How much do population-based training and exemplar-level curation improve task success rate? 3) How do our approaches compare to alternatives leveraging task-specific knowledge or hierarchical algorithms?

6.1. Experimental Setup

Benchmark Tasks We evaluate on three benchmarks: **ALFWorld (ALF)** (Shridhar et al., 2020), a text-based environment for navigation and object manipulation; **InterCode-SQL (IC-SQL)** (Yang et al., 2023), an interactive coding environment for SQL query generation; and **Wordcraft (WC)** (Jiang et al., 2020), a simplified adaptation of Little Alchemy requiring compositional reasoning.

Methods Compared Our methods include:

- **Fixed-DB**: Baseline agent with fixed database of human-provided examples
- **Traj-BS**: Simple progressive accumulation approach
- **Traj-BS+DB-Cur**: Database-level trajectory curation
- **Traj-BS+EX-Cur**: Exemplar-level trajectory curation
- **Traj-BS+DB+EX-Cur**: Both database-level and exemplar-level curation

We compare to hierarchical designs: **Autoguide** (Fu et al., 2024) and **AutoManual** (Chen et al., 2024). Unless otherwise specified, we use GPT-4o-mini and report success rates averaged over five random seeds.

6.2. Traj-BS Results

Tab. 1 presents success rates for our methods. The performance of Traj-BS generally improves with more training tasks but exhibits diminishing returns—most gains occur within the first 25% of added tasks. This efficiency decline occurs because each new example is retrieved less frequently as the database grows, a pattern consistent with

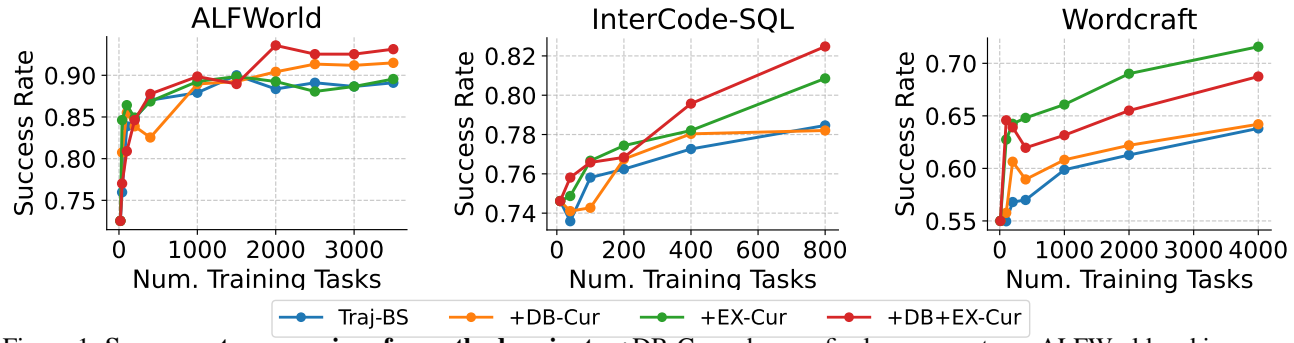


Figure 1: **Success rate comparison for method variants.** +DB-Cur enhances final success rate on ALFWorld and improves success for smaller DB sizes on all benchmarks. +EX-Cur delivers gains on InterCode-SQL and Wordcraft. The combination delivers the largest gains on ALFWorld and InterCode-SQL.

METHOD	ALF	IC-SQL	WC
FIXED-DB	0.73±0.02	0.75±0.01	0.55±0.03
TRAJ-BS	0.89±0.01	0.79±0.01	0.64±0.03
+DB-CUR	0.91±0.01	0.78±0.01	0.64±0.01
+EX-CUR	0.90±0.02	0.81±0.01	0.72±0.02
+DB+EX-CUR	0.93±0.03	0.82±0.01	0.69±0.01

Table 1: **Success rates across benchmarks.** Traj-BS outperforms Fixed-DB consistently. +DB-Cur++EX-Cur yields best results on ALF and IC-SQL.

METHOD	LLM(S)	TRAIN TASKS	ALF
AUTOGUIDE	GPT-3.5-TURBO + GPT-4-TURBO	100	0.79*
AUTOMANUAL	GPT-4O-MINI	36	0.72±0.01
	GPT-4-TURBO + GPT-4O-MINI	36	0.91±0.01
FIXED-DB	GPT-4O-MINI	0	0.73±0.05
	GPT-4O	0	0.88±0.02
+DB+EX-CUR	GPT-4O-MINI	100	0.81±0.02
	GPT-4O-MINI	3500	0.93±0.03

Table 2: **ALFWorld results.** Traj-BS+DB+EX-Cur achieves a 20-point gain over Fixed-DB, outperforming Automanual despite stronger LLMs and hand-tuning. *From original papers.

findings from Bertsch et al. (2024) and Agarwal et al. (2024). We observe performance variability both across trials and within individual trials (see App. E).

Fig. 1 illustrates how our curation methods improve upon Traj-BS. +DB-Cur enhances final success rates on ALFWorld and shows improvements for smaller database sizes across all benchmarks. +EX-Cur yields improvements on InterCode-SQL and Wordcraft. The combined +DB+EX-Cur achieves the best final success rates on ALFWorld (0.93) and InterCode-SQL (0.82).

METHOD	ALF	IC-SQL	WC
TRAJ-BS	0.89±0.01	0.79±0.01	0.64±0.03
+DB+EX-CUR	0.93±0.03	0.82±0.01	0.69±0.01
FIXED-DB@1	0.73±0.03	0.75±0.01	0.55±0.03
FIXED-DB@2	0.87±0.02	0.78±0.03	0.62±0.02
FIXED-DB@3	0.92±0.02	0.80±0.03	0.64±0.02
FIXED-DB@4	0.94±0.02	0.81±0.02	0.66±0.02
FIXED-DB@5	0.96±0.02	0.82±0.03	0.72±0.02

Table 3: **Pass@k comparison.** Traj-BS matches Fixed-DB@2–3; +DB+EX-Cur approaches Fixed-DB@5.

6.3. Contextualizing Performance Boosts

Our trajectory bootstrapping approaches achieve success rate improvements equivalent to multiple task attempts at test time (Tab. 3). Using a single attempt, Traj-BS achieves success comparable to Fixed-DB@2 or Fixed-DB@3 across all benchmarks, while +DB+EX-Cur performs nearly at the level of Fixed-DB@4 on ALFWorld and Fixed-DB@5 on InterCode-SQL and Wordcraft.

On ALFWorld, Traj-BS+DB+EX-Cur yields a 20-point success rate boost over Fixed-DB, exceeding the 15-point improvement from upgrading Fixed-DB to a more powerful LLM (Tab. 2). Our method with GPT-4o-mini (0.93) outperforms Automanual using more powerful LLMs (0.91). Given 100 training tasks, our approach is more effective (0.81) than Autoguide (0.79) despite using a weaker LLM.

We also extended Traj-BS for agent diagnostics by training a success prediction classifier (reaching AUROC of 0.77 for InterCode-SQL and 0.71 for Wordcraft), and for fine-tuning (outperforming our in-context approach on two benchmarks). See App. H and App. G for details.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Agarwal, R., Singh, A., Zhang, L., Bohnet, B., Rosias, L., Chan, S., Zhang, B., Anand, A., Abbas, Z., Nova, A., et al. Many-shot in-context learning. *Advances in Neural Information Processing Systems*, 37:76930–76966, 2024.

Akyürek, E., Schuurmans, D., Andreas, J., Ma, T., and Zhou, D. What learning algorithm is in-context learning? investigations with linear models. *arXiv preprint arXiv:2211.15661*, 2022.

Bai, Y., Jones, A., Ndousse, K., Askell, A., Chen, A., DasSarma, N., Drain, D., Fort, S., Ganguli, D., Henighan, T., et al. Training a helpful and harmless assistant with reinforcement learning from human feedback. *arXiv preprint arXiv:2204.05862*, 2022.

Bertsch, A., Ivgi, M., Alon, U., Berant, J., Gormley, M. R., and Neubig, G. In-context learning with long-context models: An in-depth exploration. *arXiv preprint arXiv:2405.00200*, 2024.

Brown, B., Juravsky, J., Ehrlich, R., Clark, R., Le, Q. V., Ré, C., and Mirhoseini, A. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787*, 2024.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33: 1877–1901, 2020.

Chen, M., Li, Y., Yang, Y., Yu, S., Lin, B., and He, X. Automanual: Generating instruction manuals by llm agents via interactive environmental learning. *arXiv preprint arXiv:2405.16247*, 2024.

Douze, M., Guzhva, A., Deng, C., Johnson, J., Szilvasy, G., Mazaré, P.-E., Lomeli, M., Hosseini, L., and Jégou, H. The faiss library. *arXiv preprint arXiv:2401.08281*, 2024.

Fu, Y., Kim, D.-K., Kim, J., Sohn, S., Logeswaran, L., Bae, K., and Lee, H. Autoguide: Automated generation and selection of context-aware guidelines for large language model agents. *arXiv preprint arXiv:2403.08978*, 2024.

Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

He, H., Yao, W., Ma, K., Yu, W., Dai, Y., Zhang, H., Lan, Z., and Yu, D. Webvoyager: Building an end-to-end web agent with large multimodal models. *arXiv preprint arXiv:2401.13919*, 2024.

Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., and Steinhardt, J. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

Jaderberg, M., Dalibard, V., Osindero, S., Czarnecki, W. M., Donahue, J., Razavi, A., Vinyals, O., Green, T., Dunning, I., Simonyan, K., et al. Population based training of neural networks. *arXiv preprint arXiv:1711.09846*, 2017.

Jaech, A., Kalai, A., Lerer, A., Richardson, A., El-Kishky, A., Low, A., Helyar, A., Madry, A., Beutel, A., Carney, A., et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

Jiang, M., Luketina, J., Nardelli, N., Minervini, P., Torr, P. H., Whiteson, S., and Rocktäschel, T. Wordcraft: An environment for benchmarking commonsense agents. *arXiv preprint arXiv:2007.09185*, 2020.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Kagaya, T., Yuan, T. J., Lou, Y., Karlekar, J., Pranata, S., Kinose, A., Oguri, K., Wick, F., and You, Y. Rap: Retrieval-augmented planning with contextual memory for multimodal llm agents. *arXiv preprint arXiv:2402.03610*, 2024.

Peters, J. and Schaal, S. Reinforcement learning by reward-weighted regression for operational space control. In *Proceedings of the 24th international conference on Machine learning*, pp. 745–750, 2007.

Rafailov, R., Sharma, A., Mitchell, E., Manning, C. D., Ermon, S., and Finn, C. Direct preference optimization: Your language model is secretly a reward model. *Advances in Neural Information Processing Systems*, 36: 53728–53741, 2023.

Reimers, N. and Gurevych, I. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019. URL http://arxiv.org/abs/1908.10084.

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

Shridhar, M., Yuan, X., Côté, M.-A., Bisk, Y., Trischler, A., and Hausknecht, M. Alfworld: Aligning text and embodied environments for interactive learning. *arXiv preprint arXiv:2010.03768*, 2020.

Song, C. H., Wu, J., Washington, C., Sadler, B. M., Chao, W.-L., and Su, Y. Llm-planner: Few-shot grounded planning for embodied agents with large language models. In *Proceedings of the IEEE/CVF international conference on computer vision*, pp. 2998–3009, 2023.

Von Oswald, J., Niklasson, E., Randazzo, E., Sacramento, J., Mordvintsev, A., Zhmoginov, A., and Vladymyrov, M. Transformers learn in-context by gradient descent. In *International Conference on Machine Learning*, pp. 35151–35174. PMLR, 2023.

Wang, E., Cassano, F., Wu, C., Bai, Y., Song, W., Nath, V., Han, Z., Hendryx, S., Yue, S., and Zhang, H. Planning in natural language improves llm search for code generation. *arXiv preprint arXiv:2409.03733*, 2024a.

Wang, X., Yang, Z., Li, L., Lu, H., Xu, Y., Lin, C.-C., Lin, K., Huang, F., and Wang, L. Scaling inference-time search with vision value model for improved visual comprehension. *arXiv preprint arXiv:2412.03704*, 2024b.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Wei, J., Wei, J., Tay, Y., Tran, D., Webson, A., Lu, Y., Chen, X., Liu, H., Huang, D., Zhou, D., et al. Larger language models do in-context learning differently. *arXiv preprint arXiv:2303.03846*, 2023.

Yang, J., Prabhakar, A., Narasimhan, K., and Yao, S. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *Advances in Neural Information Processing Systems*, 36:23826–23854, 2023.

Yang, K., Liu, Y., Chaudhary, S., Fakoor, R., Chaudhari, P., Karypis, G., and Rangwala, H. Agentoccam: A simple yet strong baseline for llm-based web agents. *arXiv preprint arXiv:2410.13825*, 2024.

Yang, Z., Qi, P., Zhang, S., Bengio, Y., Cohen, W. W., Salakhutdinov, R., and Manning, C. D. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. *arXiv preprint arXiv:1809.09600*, 2018.

Yao, S., Chen, H., Yang, J., and Narasimhan, K. Webshop: Towards scalable real-world web interaction with grounded language agents. *Advances in Neural Information Processing Systems*, 35:20744–20757, 2022.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Zhao, A., Huang, D., Xu, Q., Lin, M., Liu, Y.-J., and Huang, G. Expel: Llm agents are experiential learners. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pp. 19632–19642, 2024.

Zhou, R., Yang, Y., Wen, M., Wen, Y., Wang, W., Xi, C., Xu, G., Yu, Y., and Zhang, W. Trad: Enhancing llm agents with step-wise thought retrieval and aligned decision. In *Proceedings of the 47th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 3–13, 2024.

A. Limitations and Broader Impacts

We make the assumption that we are given a few human-provided examples at the start of the database construction process–an assumption standard in the Agentic literature (Yao et al., 2023; Zhao et al., 2024; Fu et al., 2024; Chen et al., 2024; Kagaya et al., 2024). In App. I, we explore the alternate setting of starting from an empty database, and look forward to further research on bootstrapping from zero human examples in the future. Our algorithm is dependent on the in-context capabilities of LLMs–so our algorithms may be less effective if using an LLM with weaker in-context capabilities than gpt-4o-mini, but may be more effective if paired with more advanced LLMs. In addition, our algorithm generally produces improvements in task success rates, but task success rates are not monotonically increasing, and we hope future work will help improve both the monotonicity and sample efficiency of our algorithm.

This work has the potential to allow LLM Agents applied to a variety of task domains to self-improve. This provides a variety of benefits in terms of task performance, but could lead to applications in the future where the agents perform reward hacking-type behavior–performing undesirable behaviors that are not captured in task success/failure. Since the behavior of our agents is controlled via in-context examples, a potential mitigation technique could be to manually inspect the databases of self-generated examples, or even use an LLM Judge to inspect the examples.

B. Note on Sequential Decision-Making Tasks

We focus on multi-step sequential decision-making tasks where agents must produce a series of actions over time based on observations of the environment. The sequential nature of these tasks introduces unique challenges for LLM agents, as they must interpret intermediate environmental feedback, maintain coherent reasoning across multiple steps, and adapt their strategy based on the evolving task state. This contrasts with one-shot generation tasks (e.g., solving math problems (Hendrycks et al., 2021), one-shot code generation (Jimenez et al., 2023)) where feedback is only available after the complete solution is provided. Our example-driven learning strategy is potentially also suitable to single-step decision-making tasks, but we focus on the multi-step setting due to its applicability to a number of agentic tasks in real-world settings (embodied agents (Song et al., 2023), browser-based tasks (He et al., 2024), etc.).

Formally, these tasks can be modeled as Partially Observable Markov Decision Processes (POMDPs), represented by the tuple $(\mathcal{S}, \mathcal{O}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma)$, where $\mathcal{S}$ denotes the underlying state space, $\mathcal{O}$ the observation space, $\mathcal{A}$ the action space, $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ defines the deterministic transition function, $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function, and $\gamma \in [0, 1]$ is the discount factor. The partial observability reflects that agents don't have direct access to the full environment state but rather receive observations that provide limited information.

Given a task goal g, an episode consists of the agent interacting with the environment for a maximum of T timesteps. At each timestep t, the agent receives an observation $o_t \in \mathcal{O}$ of the current state, takes an action $a_t \in \mathcal{A}$, and the environment transitions to the next state according to the transition function $\mathcal{T}$. In our setting, we specifically consider sparse-reward environments where success is only determined at the end of an episode—the agent receives $\mathcal{R} = 1$ for successful task completion and $\mathcal{R} = 0$ otherwise. This is a standard setting in prior agentic work (Yao et al., 2023; Zhao et al., 2024; Fu et al., 2024; Chen et al., 2024).

C. Key Agent Details

In Sec. 2, we establish an agent design that enables it to learn in-context from its own self-collected experiences. Here, we elaborate on a few key design decisions in our agent design:

- **Standardized prompts**: we use the same simple, task-agnostic prompt templates for all tasks, rather than writing new prompts per task. These prompts are in App. D. Alternate appraoches incorporate domain-specific information into their prompts–we discuss these approaches in Appendices J.1 and K.
- **Two-level retrieval**: We retrieve trajectories at both trajectory level (for planning) and state level (for reasoning and action selection), enabling the agent to leverage both strategic patterns and situation-specific techniques. Database $\mathcal{D}$ contains self-collected trajectories, and retrieval is performed at the trajectory level for the initial plan p, and in the state-level observation-reasoning-action loop for both r_t and a_t.
- **Multi-key retrieval**: All retrieval is performed by KNN, with similarity metric defined as the average of cosine similarities across the specified 'key' variables. For instance, in Line 3 of Alg. 1, we retrieve from $\mathcal{D}$ using two keys:

Algorithm 4 Multi-Key Retrieval

```
1:  procedure MULTIKEYRETRIEVAL(𝒟, traj_keys, state_key, query, k, window_size)
2:      similarities ← []
3:      for each trajectory τ in 𝒟 do
4:          sim ← 0
5:          for each key in traj_keys do
6:              sim ← sim + COSINESIMILARITY(query[key], τ[key])
7:          sim ← sim/|traj_keys|                          {Average similarity across keys}
8:          similarities.APPEND(sim, τ)
9:      similar_trajectories ← TOPK(similarities, k)
10:     if state_key ≠ None then
11:         windowed_results ← []
12:         for each trajectory τ in similar_trajectories do
13:             state_similarities ← []
14:             for each state s in τ.states do
15:                 state_sim ← COSINESIMILARITY(query[state_key], s[state_key])
16:                 state_similarities.APPEND(state_sim, s, INDEX(s))
17:             _, most_similar_state, idx ← MAX(state_similarities)
18:             start ← max(0, idx − ⌊window_size/2⌋)
19:             end ← min(|τ.states|, idx + ⌈window_size/2⌉)
20:             windowed_results.APPEND(τ.states[start:end])
21:         return windowed_results
22:     else
23:         return similar_trajectories
```

goal g and plan p. We return similar trajectories based off the average of the cosine similarities of goals and plans when comparing each trajectory to the current trajectory. When doing state-level retrieval (Lines 7 and 10), we additionally find the most similar states within the selected trajectories via state-level key o_t or r_t, then return a window of states around the most similar state. This is similar to the retrieval scheme in (Kagaya et al., 2024). See detailed pseudocode for retrieval in Alg. 4.

- **Thought-based retrieval**: For the first step of a trajectory, we retrieve using the trajectory-level keys (g,p) as well as the current observation o_1 (Alg. 1, line 6)–but for all subsequent steps we use reasoning r_t as a key instead of observation o_t (Alg. 1, line 9). This approach, inspired by Zhou et al. (2024), enables generalization across trajectories with similar reasoning, and similarity across natural-language r_t can be handled by generic embedding functions more easily than potentially bespoke observations o_t. By retrieving at every step, we aim to retrieve the most relevant trajectories for each decision.

- **Generic embedding mechanism**: Since g, p, and r_t are all natural-language strings, we employ standard embeddings (all-MiniLM-L6-v2 (Reimers & Gurevych, 2019)) that generalize across domains without task-specific engineering.

D. Additional Implementation Details

D.1. Hyperparameters

Unless otherwise specified, we use GPT-4o-mini as our base LLM (temperature 0.1). For Fixed-DB and all Traj-BS agents, we retrieve the top-k most similar trajectories at each decision step ($k = 6$ for ALFWorld and InterCode-SQL, 10 for Wordcraft). We initialize each database with a small human-provided example set (18 for ALFWorld, 10 for InterCode-SQL, 4 for Wordcraft). With +DB-Cur, we maintain $N = 5$ database instances with curation every time the database size is doubled, starting with a minimum size of ten trajectories. We report success rates averaged over five random seeds. The standard deviation of the success rates is also reported. By default, we report success rate given the database at the end of the training process.

D.2. Prompt Templates

Across all benchmarks, we use standardized prompt templates for the core components of our retrieval-based ReAct agent. The same templates were used across all benchmarks with no task-specific modifications. These templates are intentionally minimalist, focusing on providing the necessary context and retrieved examples while avoiding task-specific prompt engineering.

The templates are included below. Across all templates, the in-context examples follow the format specified in the prompt itself (for plan, the in-context examples are of form "goal,plan", etc):

Plan:

```
system_prompt: 'You are an expert at generating high-level plans of actions to achieve a goal.\n Here is your action space: {action_space}.\n Here are some examples of goal,plan from episodes that successfully achieved similar goals: {examples}'
user_prompt: 'goal: {goal}\n plan: '
```

Reason:

```
system_prompt: 'You are an expert at reasoning about the most appropriate action to take towards achieving a goal.\n Here is your action space: {action_space}.\n Here are some examples of goal,plan,observation,reasoning,action from episodes that successfully achieved similar goals: {examples}'
user_prompt: 'goal: {goal}\n plan: {plan}\n trajectory: {trajectory}\n reasoning: '
```

Act:

```
system_prompt: 'You are an agent in an environment. Given the current observation, you must select an action to take towards achieving the goal: {self.goal}.\n Here is your action space: {action_space}.\n Here are some examples of goal,plan,observation,reasoning,action from episodes that successfully achieved similar goals: {examples}'
user_prompt: 'goal: {goal}\n plan: {plan}\n trajectory: {trajectory}\n action: '
```

D.3. Retrieval Implementation

For all retrieval steps, we implement hybrid search across all the desired retrieval keys–ex. goal, plan, observation, reasoning. We return the top-k examples by averaged distance across each of the keys. We implement a sliding window approach for state-level retrieval to enhance contextual relevance–we include the surrounding context (preceding and following states) up to a window of 5 steps to provide coherent episode fragments.

The retrieval mechanism is implemented using FAISS (Douze et al., 2024) for efficient similarity search as the database grows. We use exact nearest neighbor search.

D.4. Population-Based Training Details

Our database-level curation approach maintains a population of 5 database instances. Each instance is initialized with the same set of human-provided exemplars. The population undergoes curation every time the database size doubles, and performance is evaluated on the tasks attempted since the previous doubling.

The replacement strategy follows standard population-based training practices: the bottom 20% of databases (based on validation performance) are replaced with copies of the top 20%.

D.5. Quality Metric Computation

For exemplar-level curation, we track the retrieval patterns of each trajectory throughout the training process. For each task, we record: 1. Which trajectories were retrieved 2. How many times each trajectory was retrieved during the solution process 3. Whether the task was successfully completed

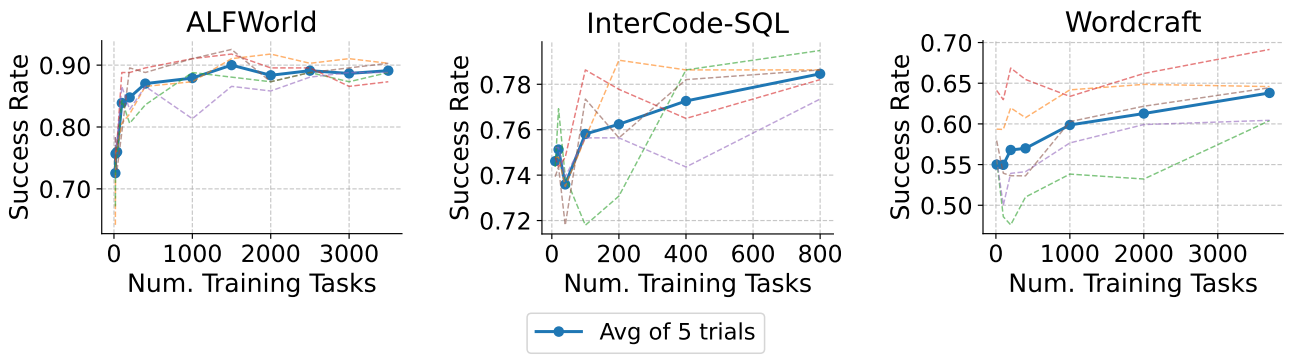


Figure 2: **Traj-BS results: success rate improves with increasing training tasks on all three benchmarks**. Individual trials (5) shown as dashed lines. All benchmarks exhibit diminishing returns as the database size increases. Trials show substantial performance variability, both within individual trials and across different trials.

After completing all training tasks, we compute the quality metric $Q(\tau)$ for each trajectory τ as:

$$Q(\tau) = \frac{\sum_{i \in \mathcal{R}(\tau)} o_i \cdot f_i(\tau)}{\sum_{i \in \mathcal{R}(\tau)} f_i(\tau)} \tag{2}$$

where $\mathcal{R}(\tau)$ is the set of tasks for which trajectory τ was retrieved, $o_i \in \{0, 1\}$ is the outcome of task i, and $f_i(\tau)$ is the retrieval frequency of trajectory τ during task i.

To ensure statistical significance, we only compute the quality metric for trajectories that were retrieved for at least 3 different tasks. For trajectories with insufficient retrieval data, we assign a neutral quality score equal to the average success rate across all tasks.

D.6. Note on planning step

Following the convention from RAP (Kagaya et al., 2024), we omit the planning step on benchmarks with short trajectory length (Intercode-SQL, Wordcraft). This planning step is valuable for maintaining long-horizon coherence on the ALFWorld benchmarks (30 steps), and is standard in prior ReAct-based agentic work (Kagaya et al., 2024; Zhao et al., 2024; Fu et al., 2024), whether the planning step is explicitly separate from reasoning, or incorporated into the first reasoning step.

E. Per-Trial Analysis

The performance of Traj-BS exhibits significant variability both across different trials and within individual trials, as shown in Fig. 2. This variance stems from two primary factors: (1) the inherent stochasticity of LLM outputs creating different initial trajectories, and (2) a compounding effect where early differences in collected examples lead to divergent solution strategies.

Cross-trial variance indicates that some random seeds produce higher-performing databases even when solving identical task sequences. This suggests the initial trajectory collection phase is critical—early successes guide the agent toward particular solution patterns that persist throughout training. Within-trial fluctuations demonstrate that certain added trajectories can temporarily degrade performance, particularly when they introduce reasoning patterns that conflict with existing examples or when they succeed through coincidence rather than robust reasoning.

F. Exemplar Quality Metric Analysis

To highlight the impact of our exemplar-level curation metric from Eq. 1, Fig. 3 compares databases built from the 'best' trajectories that are the most empirically effective in-context examples versus the least effective trajectories, as determined by Equation 1 in Sec. 5.3. The 'best' curve is identical to +EX-Cur, while the 'worst' curve selects the bottom-1 trajectory instead of top-1 in Alg. 3, line 7. Using the database of high-quality examples yields a higher success rate across all database sizes for ALFWorld and Wordcraft, and for smaller database sizes for InterCode-SQL.

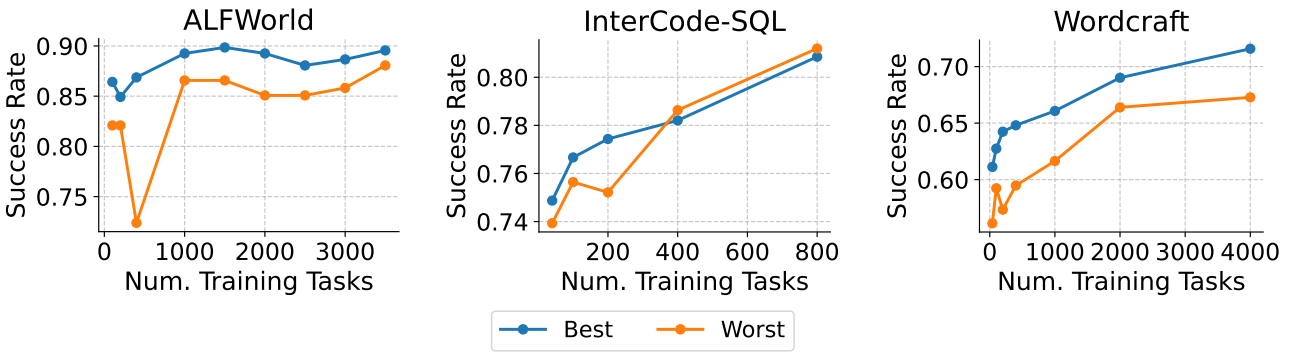


Figure 3: **The 'best' bootstrapped trajectories compared to the 'worst'**. Databases constructed from the highest-quality successful trajectory per task, as measured by Eq. 1, outperform databases built from the lowest-quality successful trajectories on both ALFWorld and Wordcraft. The 'best' curve is identical to +EX-Cur, while the 'worst' curve selects the bottom-1 trajectory instead of top-1 in Alg. 3, line 7.

Method	ALFWorld	InterCode-SQL	Wordcraft
Traj-BS+DB+EX-Cur	0.93±0.03	**0.82**±0.01	0.69±0.01
ReAct-Finetune	**0.96**±0.01	0.79±0.01	**0.74**±0.01

Table 4: **Trained on the same data, fine-tuned agent ReAct-Finetune is competitive with our best in-context approach.** This suggests that our self-collected data is effective not only for in-context prompting but also for creating fine-tuned agents. All values are averages over 5 trials.

G. Can Self-Collected Examples Improve a Fine-Tuned LLM Agent?

We have shown that self-collected databases improve the performance of in-context LLM agents. Here, we test whether the same data can also benefit fine-tuning.

Using the OpenAI fine-tuning API, we fine-tune GPT-4o-mini on each benchmark using data from our best-performing database construction method: Traj-BS+DB+EX-Cur, collected over the full training set. We fine-tune using a simple ReAct-format prompt:

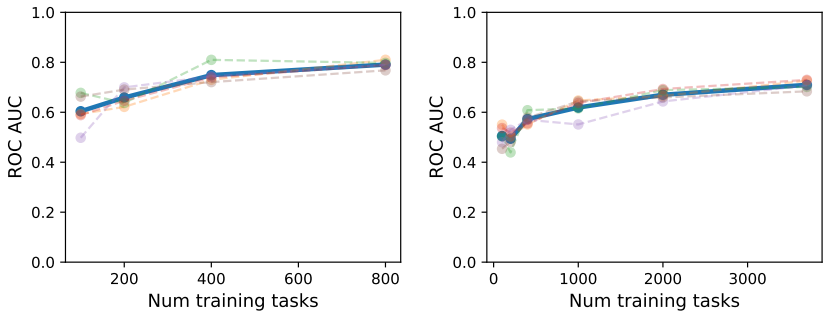
```
{
'system':'You are a ReAct agent that helps users accomplish tasks. Given a goal, you will receive observations about the environment and respond with your reasoning and actions. For each observation, first think through the problem step by step (Thought), then decide on an action (Action). Your actions should be clear, concise, and directly executable in the environment.',
'user':'Goal: {goal} \n Initial observation: {observations[0]}',
'assistant':'Thought: {reasoning[i]}\nAction: {action[i]}',
'user':'Observation: {observations[i+1]}',
...
}
```

We refer to the resulting fine-tuned model as ReAct-Finetune. To run the agent, we prompt it with a goal and initial observation, then alternate assistant messages (for reasoning and action) with user messages (for new observations).

Tab. 4 shows that ReAct-Finetune slightly outperforms the in-context agent on ALFWorld (0.96 vs. 0.93) and Wordcraft (0.74 vs. 0.69), while performing slightly worse on InterCode-SQL (0.79 vs. 0.82). These results suggest that self-collected examples are effective not only for in-context prompting but also for creating competitive fine-tuned agents.

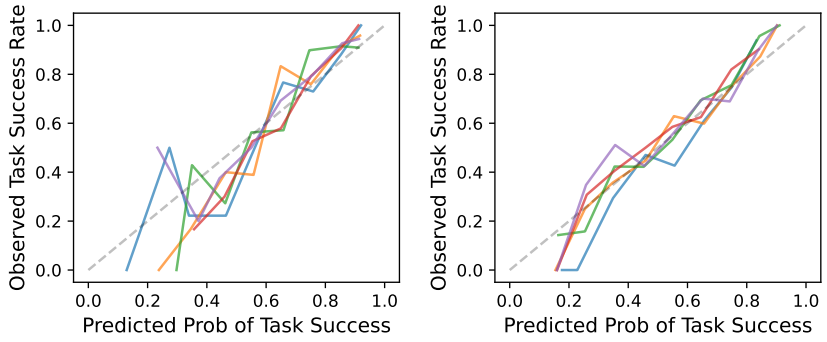
H. Can We Predict Agent Success Rates

We have shown that increasing the number of self-collected examples improves agent performance. Here, we test whether the same examples can also predict performance on new tasks.



(a) InterCode-SQL (b) Wordcraft

Figure 4: **AUROC of success prediction improves with more self-collected examples.** Performance continues to rise with increasing database size.



(a) InterCode-SQL (b) Wordcraft

Figure 5: **Predicted probabilities are well-calibrated.** For both benchmarks, predicted and empirical success rates generally align.

On InterCode-SQL and Wordcraft, task difficulty is partly observable from the goal g and initial observation o_1. For InterCode-SQL, g is a natural-language query. For Wordcraft, g is the desired element and o_1 specifies available crafting elements. In contrast, ALFWorld task difficulty depends heavily on scene layout, which g and o_1 do not reveal. We therefore exclude ALFWorld from this analysis.

We use the same embedding model as in retrieval (all-MiniLM-L6-v2 (Reimers & Gurevych, 2019)) to encode the concatenated string $[g; o_1]$. We train a calibrated Random Forest classifier to predict task success/failure, calibrating its outputs via 5-fold cross-validation with a learned sigmoid function. For each of 5 independent Traj-BS trials, we evaluate (1) the classifier's AUROC on held-out tasks, and (2) its calibration.

As shown in Fig. 4, prediction performance improves as the database grows. For InterCode-SQL, AUROC rises from 0.60 (100 tasks) to 0.77 (800 tasks). Wordcraft shows a similar trend, improving from 0.50 (100 tasks) to 0.71 (4000 tasks). In both cases, predictive accuracy increases alongside task performance.

Fig. 5 shows the calibration of the final classifiers (trained on all available training tasks). Predicted success probabilities closely match observed success rates, indicating well-calibrated models.

I. Is it Possible to Bootstrap an Agent Without Initial Hand-Crafted Examples?

Providing a small number of hand-crafted in-context examples is standard practice in the LLM agent literature (Yao et al., 2023; Zhao et al., 2024; Fu et al., 2024; Chen et al., 2024; Kagaya et al., 2024). However, what if we initialized Traj-BS with an empty database? In order to understand the value of the initial human-provided examples, we test Traj-Boostrap with and without the initial human-provided examples on Wordcraft. We refer to the variant initialized with an empty database

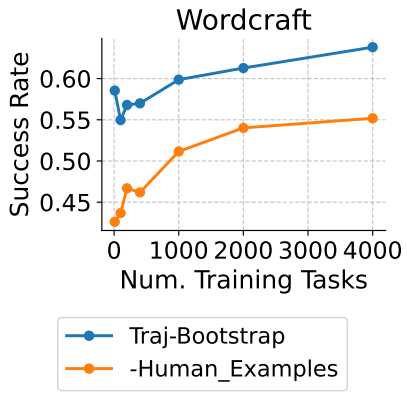


Figure 6: **Ablating the value of initial human-provided examples, Wordcraft.** Traj-BS, initialized by default with a database of 5 human-provided trajectories for Wordcraft, achieves better performance with these starting examples than when initialized from an empty database (-Human-Examples). Performance still scales with database size for -Human-Examples–but in this case fails to reach the performance achieved via 5 human-provided examples, even after self-collecting trajectories on 4000 training tasks.

as -Human-Examples. Traj-BS, initialized by default with a database of 5 human-provided trajectories for Wordcraft, achieves better performance with these starting examples than when initialized from an empty database (-Human-Examples). Performance still scales with database size for -Human-Examples–but in this case fails to reach the performance achieved via 5 human-provided examples, even after self-collecting trajectories on 4000 training tasks. On at least this one task, the initial human-provided trajectories shaped the reasoning and action patterns of the agent in a way that boosted the continual database construction process. We leave exploration of hand-crafting these in-context examples to future work.

J. Key Details of Prior Agentic Approaches

J.1. How does Automanual Leverage Hand-Crafted Information

Rather than learning from self-collected examples, an alternate approach to agent construction is to leverage practitioner domain knowledge. Beyond implementing both a hierarchical learning system and code-based action spaces, Automanual (Chen et al., 2024) incorporates domain knowledge about the ALFWorld task into multiple components of the algorithm. In this section we include some code from the official Automanual GitHub to illustrate.

Observation spaces : Automanual uses a modified observation space that enhances the ALFWorld string by adding two critical pieces of information: 1) The current location of the agent, 2) What the agent is currently holding. Both of these pieces of information typically have to be deduced from the trajectory of previous observations and actions, but Automanual tracks them explicitly:

```
if "Nothing happens" not in observation:
    self.last_obs = observation
    if "go to" in script:
        self.cur_loc = re.search(r'go to (\S+)', script).group(1)
        self.cur_loc_info = observation
    if "open" in script or "close" in script:
        self.cur_loc_info = observation
    if "take" in script:
        self.holding = re.search(r"(?<=take\s)(.*?)(?=\sfrom)", script).group(1)
        self.cur_loc_info = ""
    if "put" in script:
        self.holding = "nothing"
        self.cur_loc_info = ""
elif "go to" in script:
    loc = re.search(r'go to (\S+)', script).group(1)
```

```
    if loc == self.cur_loc:
        observation = self.cur_loc_info
observation += f" You are at {self.cur_loc} and holding {self.holding}."
```

Action spaces : in ALFWorld, any task typically involves three main components: 1) Searching for an object, 2) Performing an action with the object (heating, cooling, cleaning, etc.), 3) Placing the object somewhere. Automanual significantly simplifies both the search and placement operations by providing multi-action helper functions within its code-based action space:

```
# Define a helper method to find object that is needed
def find_object(agent, recep_to_check, object_name):
    for receptacle in recep_to_check:
        observation = agent.go_to(receptacle)
        # Check if we need to open the receptacle. If we do, open it.
        if 'closed' in observation:
            observation = agent.open(receptacle)
        # Check if the object is in/on the receptacle.
        if object_name in observation:
            object_ids = get_object_with_id(observation, object_name)
            return object_ids, receptacle
    return None, None

# Define a helper method to put object in/on the target receptacle
def go_to_put_object(agent, target_receptacle, object_id):
    observation = agent.go_to(target_receptacle)
    # check if target_receptacle is closed. If so, open it.
    if 'closed' in observation:
        observation = agent.open(target_receptacle)
    observation = agent.put_in_or_on(object_id, target_receptacle)
    return observation
```

J.2. A note on training and test sets

The distinction between how different techniques leverage data is crucial in understanding the generalization capabilities of LLM agents. We can categorize existing approaches based on how they treat training and test data:

Single-Task Optimization : Some approaches focus exclusively on improving performance on a single task instance without concern for generalization. For example, Shinn et al. (2023) leverages feedback from failed attempts to incrementally improve performance on the same task. Similarly, search methods (Brown et al., 2024; Wang et al., 2024b) expand the solution space for a specific problem instance. While these approaches can solve individual tasks, they don't transfer knowledge across different problems, essentially 'overfitting' to a single instance.

Mixed Train-Test Evaluation : Some recent work blurs training and test boundaries. For instance, RAP (Kagaya et al., 2024) makes multiple passes over the same dataset, allowing the system to 'learn' from some test examples before evaluating on others within the same set. This approach does not assess true generalization capability, as the model has indirect exposure to the test distribution during its learning phase.

Full Train-Test Separation : Several papers maintain a clear separation between training and test data: 1) ExpeL (Zhao et al., 2024) extracts general rules from a training set of trajectories and applies them to entirely separate test tasks, 2) AutoGuide (Fu et al., 2024) generates contextual guidelines from training experiences that are evaluated on distinct test scenarios. 3) AutoManual (Chen et al., 2024) constructs hierarchical 'manuals' from training interactions that are then applied to novel test tasks.

Our approach similarly ensures that trajectories used for database construction come exclusively from designated training tasks, with evaluation conducted on a separate set of test tasks never seen during the database construction phase. This separation is essential for validating that the knowledge captured by the agent generalizes to new problems rather than memorizing specific solutions.

Method	Intercode-SQL Success Rate
GameSQL	0.73
GameSQL+Cheat	**0.84**
Fixed-DB	0.74
Traj-BS	0.79
+DB-Cur	0.78
+EX-Cur	0.81
+DB+EX-Cur	**0.82**

Table 5: **Comparison of agent success rates on InterCode-SQL: contextualizing the performance of Traj-Boostrap.** Without cheats, the hand-crafted GameSQL agent (0.73) performs comparably to Fixed-DB (0.74). With handicap access to the database schema, GameSQL+Cheat (0.84) slightly outperforms +DB+EX-Cur (0.82). The boost from our database-construction techniques nearly matches the boost from providing the GameSQL agent with access to privileged database schema information.

K. Comparison to Hand-Crafted InterCode-SQL Agent

We further contextualize the performance of Traj-BS by comparing to two hand-crafted agents on InterCode-SQL. The Intercode-SQL paper (Yang et al., 2023) provides a hand-crafted agent, GameSQL to solve the task, and optionally provides the agent with a 'handicap'–giving the agent information on all relevant parts of the database schema. We denote the assisted version as GameSQL+Cheat. Neither agent provides in-context examples, and both share a bespoke, hand-crafted prompt (see App. N).

As seen in Tab. 5, Fixed-DB performs similarly to GameSQL (0.74 vs 0.73), and the performance of our best method, +DB+EX-Cur, approaches the performance of GameSQL+Cheat (0.82 vs 0.84). Therefore, our database-construction techniques lift the performance of a generic ReAct-style agent nearly as much as the lift provided to the hand-crafted agent via 'handicap' access to the database schema.

L. Benchmark Details

L.1. ALFWorld

ALFWorld (Shridhar et al., 2020) is a text-based environment that aligns with embodied tasks, allowing agents to navigate and manipulate objects through textual commands. We use the standard ALFWorld benchmark consisting of 3500 training tasks and 134 out-of-distribution test tasks across 6 task categories:

- Pick & Place: Find and move objects to specified locations
- Clean & Place: Find, clean, and place objects
- Heat & Place: Find, heat, and place objects
- Cool & Place: Find, cool, and place objects
- Pick Two & Place: Find and move two objects to a specified location
- Look at Object: Find an object and examine it under light

Following (Kagaya et al., 2024), for the ALFWorld benchmark we perform similarity search over task categories in addition to the other retrieval keys (goal, plan, observation, action). We do this to follow the convention in this prior work.

For our initial human-provided exemplars, we used the 18 successful trajectories (3 per task category) provided by Zhao et al. (2024). These trajectories were used to initialize all database instances.

The success criteria for ALFWorld tasks are defined by the environment and require the agent to satisfy all conditions specified in the goal. For example, in a 'Heat & Place' task, the agent must find the target object, place it in the microwave,

turn on the microwave, and finally place the heated object at the specified destination. Both Autoguide and Automanual allow 50 actions for task completion–but choosing to employ "reasoning" counts as an action. Since we force our agent to reason at every step, we allow our agents (Fixed-DB, Traj-BS and variants) only 30 steps for task completion (on Autoguide and Automanual, the agent does not reason in practice at most steps ex. in a search procedure).

For this benchmark, we do not provide an action space string to the LLM, relying purely on the in-context examples to communicate the action space.

L.2. InterCode-SQL

InterCode-SQL (Yang et al., 2023) is an interactive coding environment for evaluating language agents' SQL programming abilities. We use a subset of the InterCode benchmark focusing on SQL query generation, built upon the Spider SQL dataset. Of the 1034 tasks in the dataset, we randomly assign 800 tasks to train and the remaining 234 tasks to test.

Each task provides a natural language query request. The agent must generate a syntactically correct SQL query that retrieves the requested information. The agent must first execute queries to understand the database schema. The environment provides feedback on syntax errors and execution results, but the agent is only allowed to submit a solution once.

The success criteria for InterCode-SQL tasks require the agent to submit a solution query within 10 steps. The environment executes the query and compares the results against a ground-truth reference.

For our initial human-provided exemplars, we collected 10 human-created trajectories for 10 randomly-selected training tasks. These trajectories were used to initialize all database instances. For all solved trajectories, we append the solution query to the goal string–since the goal of the task is to 'discover' this query through interacting with the SQL database.

We used the following action space string for InterCode-SQL:

```
Your action space is outputting valid mysql commands to solve the sql task.
You will be evaluated on the Latest Standard Output.
If you believe the latest observation is the final answer, you can complete the task by
    running 'submit' by itself.
You have 10 iterations to solve the task.
Follow the syntax and logical flow from the provided examples exactly.
```

L.3. Wordcraft

Wordcraft (Jiang et al., 2020) is a simplified adaptation of the game Little Alchemy, where agents must combine elements to create new elements through multi-step processes. We randomly select 4000 training tasks and 500 test tasks from the subset of tasks requiring up to 2 steps to solve, with the train-test split separating the tasks into disjoint sets of goal elements.

The agent starts with a set of elements and must discover combinations that creates a particular target element specified in the goal. The environment provides feedback on successful combinations and updates the available elements accordingly.

The success criteria for Wordcraft tasks require the agent to create the target element within 4 steps, while the minimum solution length is up to 2 steps.

For our initial human-provided exemplars, we collected 4 human-annotated trajectories from randomly-selecting training tasks. These trajectories were used to initialize all database instances. We collected fewer initial trajectories for Wordcraft than for InterCode-SQL (4 vs 10) since Wordcraft is a slightly simpler task, requiring up to 4 steps for task completion while InterCode-SQL requires up to 10.

We used the following action space string for Wordcraft:

```
Output strings with the names of the two entities we would like to combine in this step.
```

L.4. Note on Benchmark Selection

We selected three sequential decision-making benchmarks that cover different reasoning challenges—**ALFWorld** (Shridhar et al., 2020) tests text-based navigation and object manipulation, **InterCode-SQL** (Yang et al., 2023) tests interactive code generation, and **Wordcraft** (Jiang et al., 2020) tests compositional reasoning.

While prior works (Zhao et al., 2024; Fu et al., 2024) test on WebShop (Yao et al., 2022), we encountered bugs in generating achievable goals on the full benchmark (confirmed by https://github.com/princeton-nlp/WebShop/issues/43) and identified tasks with incorrect rewards.

We excluded QA benchmarks (HotPotQA (Yang et al., 2018), etc.) because performance depends on information retriever quality and LLM self-evaluation efficacy, two factors that would confound our study of LLM Self-Improvement. We plan to test our algorithms on QA benchmarks in future work.

M. Computational Resources

All experiments were conducted using the following computational resources:

- 1 NVIDIA A5000 GPU (24GB memory) for embedding computation
- 64GB RAM

The majority of computation was spent on OpenAI API calls for the LLM-based decision-making. Database operations including embedding computation, storage, and retrieval accounted for less than 5% of the total computation time.

For embedding computations, we used all-MiniLM-L6-v2 (Reimers & Gurevych, 2019).

For LLM inference, we used the OpenAI API for GPT-4o-mini, which required approximately:

- 2,000,000 API calls for ALFWorld
- 200,000 API calls for InterCode-SQL
- 500,000 API calls for Wordcraft

The total cost of API usage was approximately $3,000 USD.

N. GameSQL Prompt

Yang et al. (2023) write this hand-crafted prompt for the GameSQL agent:

```
'''
{self.language}Env` is a multi-turn game that tests your ability to write
a {self.language} command that produces an output corresponding to a natural language
 ↪  query.

## GAME DESCRIPTION
At the start of this game, you are given a natural language query describing some
desired output (i.e. "Find the first name of a student who have both cat and dog
 ↪  pets").
Aside from the natural language query, you have no information about the tables you
 ↪  have access to.

The game will be played in a series of turns. Each turn, you can submit a
 ↪  {self.language} command.
You will then get a response detailing the output of your {self.language} query along
 ↪  with a reward
that tells you how close your {self.language} command is to the correct answer.

The goal of this game is to write a {self.language} command that gets a reward of 1.
 ↪  The game will automatically
terminate once you get a reward of 1.

## INPUT DESCRIPTION
Each turn, you can submit a {self.language} command. Your {self.language} command
 ↪  should be formatted as follows:
```

```
```{self.language}
Your {self.language} code here
```

Your {self.language} command can help you do one of two things:
1. Learn more about the tables you have access to
2. Execute {self.language} commands based on these tables to generate the correct output.

OUTPUT DESCRIPTION
Given your {self.language} command input, `{self.language}Env` will then give back output formatted as follows:

Output: <string>
Reward: <decimal value between 0 and 1>

The output is a string displaying the result from executing your {self.language} query.
The reward is a decimal value between 0 and 1.

REWARD DESCRIPTION
The reward should be interpreted as a ratio. It tells you how many rows your {self.language}
command outputted correctly compared to the correct answer.

RULES
1. Do NOT ask questions. Your commands are fed directly into a SQL compiler.

STRATEGY
You are free to play as many turns of the game as you'd like to inspect tables
and develop your {self.language} command.

The best strategy for this game is to first write {self.language} commands that help you learn
about the tables that you have access to. For instance, in a SQL environment, you might use `SHOW TABLES`
and `DESC <table name>` to learn more about the tables you have access to.

Once you have a good understanding of the tables, you should then write {self.language} commands
that would answer the natural language query using the tables you have access to.
'''
```