Hi! I'm Vishnu Sarukkai, a PhD student advised by Kayvon Fatahalian. Today, I'm going to talk about how, when building effective LLM Agents, self-generated experience can be as effective as more familiar approaches such as test-time scaling, using a better model, prompt engineering, etc.

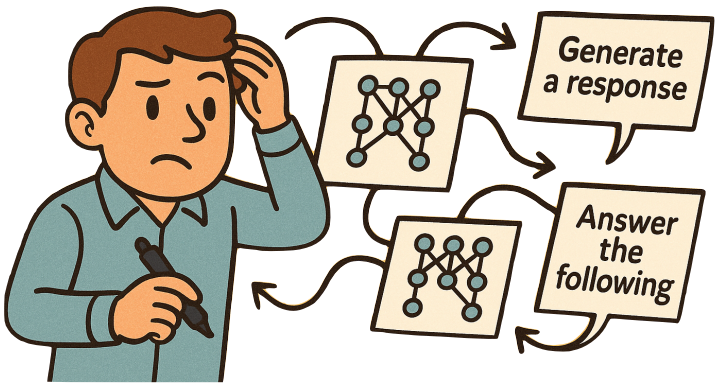
How can we make LLM Agents better?

Test-time scaling? Bigger model?

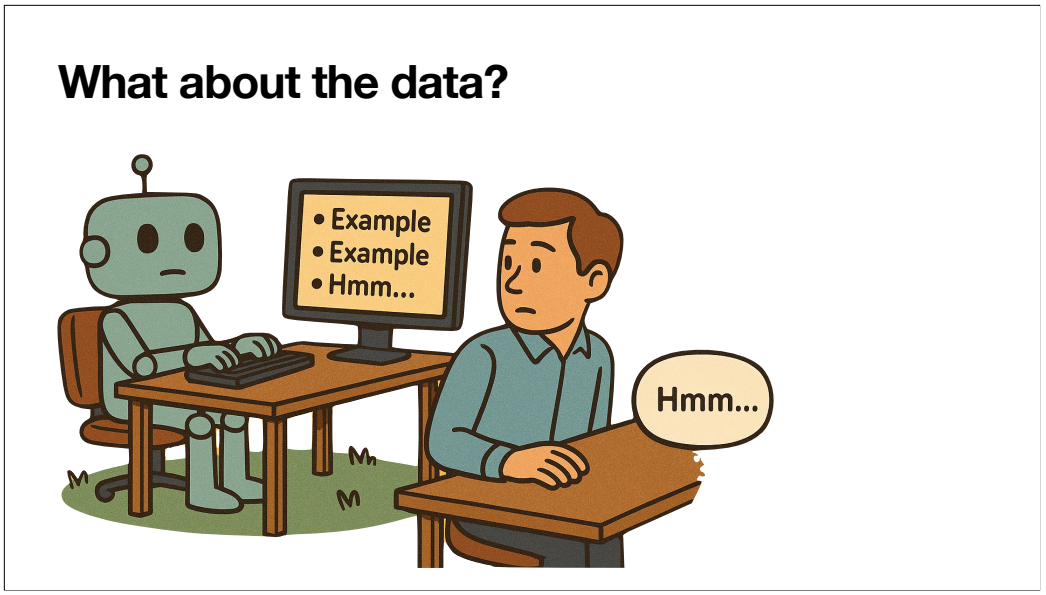
So for most of you, if you wanted to make an agent better, you might go to use a more expensive model, and pay the cost. You might adopt some of the ideas of test-time compute you've heard today, and pay the cost…

How can we make LLM Agents better?

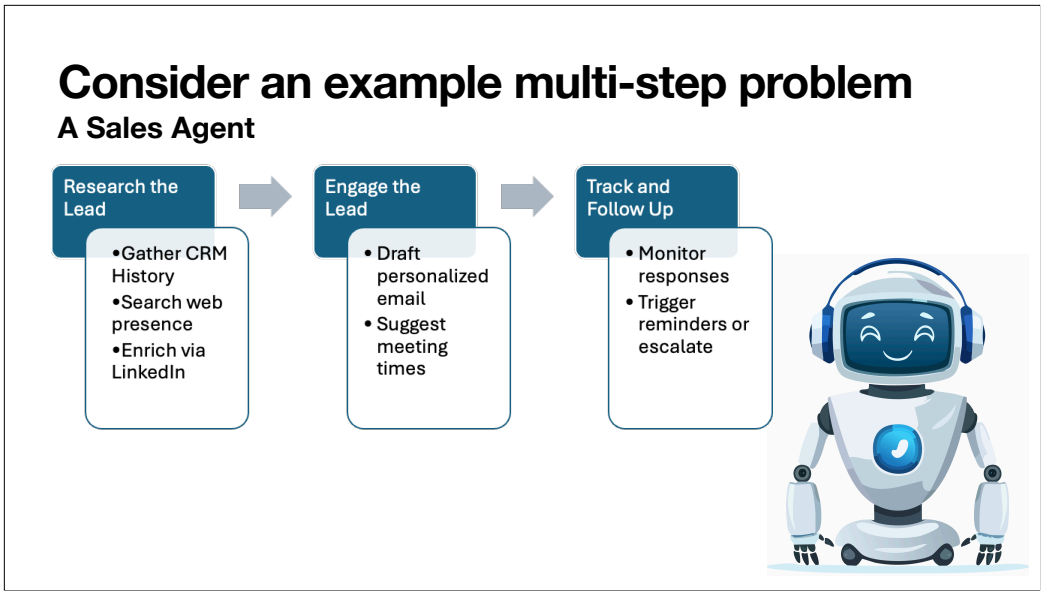
Try out architectures/prompts?



…Or you might spend a bunch of time tweaking various agent modules and hope that something will work…



In a world where everyone talks about data, it’s pretty interesting that we don’t talk too often about data for LLM agents. Maybe that’s because people want to deploy agents for use cases where there isn’t any data yet.
But deployed Agents are generating data all the time. And sometimes there’s signal to know if they’re successful.



For example, consider an LLM-based sales agent trying to convert a lead into a paying customer. Given a lead, it has to do background research on the lead—via a CRM, the web, and LinkedIn, then engage the lead via personalized outreach, then continue to follow up with the lead until conversion. This is a challenging, multi-step problem—with feedback you can collect on whether the agent is successful at converting the lead, or at least scheduling a meeting! A deployed sales agent might collect hundreds or thousands of trajectories daily.

Let’s use that data to make our agents better!

At this point, you might be thinking either:

1. Let’s throw all that data in a database, and use it in-context via a RAG-based agent.
2. Let’s throw all that data in a database, and then carry out some type of LLM finetuning run every day/week?

At this point, you might be thinking either:
1. Let’s throw all that data in a database, and use it in-context via a RAG-based agent.
2. Let’s throw all that data in a database, and then carry out some type of LLM finetuning run every day/week?
For now, let’s focus on in-context agents for a bit and describe how to do the first case well. And I think it’s really powerful since you don’t have to retrain—and in-context agents are cheap and easy to prototype.

Let’s focus on in-context ReAct agents

No prompt engineering, test-time compute, …

```
def react_agent(task_goal, db, env):
   plan = make_plan(task_goal)

   while (!env.done)
      obs = env.getobs()
      r = reason_for_action(task_goal, plan, obs)
      action = choose_action(task_goal, plan, obs, r)
      env.step(action)
```

I want to do this as generally as possible—so I don’t want to do any prompt engineering or sophisticated agent design. So I’m going to take an agent architecture is simple, but representative of common practice. It's called a ReAct agent.…

Let's focus on in-context ReAct agents

No prompt engineering, test-time compute, …

```
def react_agent(task_goal, db, env):
   plan = make_plan(task_goal)

   while (!env.done)
      obs = env.getobs()
      r = reason_for_action(task_goal, plan, obs)
      action = choose_action(task_goal, plan, obs, r)
      env.step(action)
```

…And as you can see in the diagram given a TASK GOAL…

Let's focus on in-context ReAct agents

No prompt engineering, test-time compute, …

```
def react_agent(task_goal, db, env):
   plan = make_plan(task_goal)

   while (!env.done)
      obs = env.getobs()
      r = reason_for_action(task_goal, plan, obs)
      action = choose_action(task_goal, plan, obs, r)
      env.step(action)
```

…it's going to make a plan…

Let's focus on in-context ReAct agents

No prompt engineering, test-time compute, …

```
def react_agent(task_goal, db, env):
   plan = make_plan(task_goal)

   while (!env.done)
      obs = env.getobs()
      r = reason_for_action(task_goal, plan, obs)
      action = choose_action(task_goal, plan, obs, r)
      env.step(action)
```

…and at every step it's going to do some basic reasoning about what it should do next…

Let's focus on in-context ReAct agents

No prompt engineering, test-time compute, …

```
def react_agent(task_goal, db, env):
   plan = make_plan(task_goal)

   while (!env.done)
      obs = env.getobs()
      r = reason_for_action(task_goal, plan, obs)
      action = choose_action(task_goal, plan, obs, r)
      env.step(action)
```

and then choose an action.

Let's focus on in-context ReAct agents

No prompt engineering, test-time compute, …

```
def react_agent(task_goal, db, env):
   plan = make_plan(task_goal)

   while (!env.done)
      obs = env.getobs()
      r = reason_for_action(task_goal, plan, obs)
      action = choose_action(task_goal, plan, obs, r)
      env.step(action)
```

Feedback from the environment gives me an observation…

Let’s focus on in-context ReAct agents

No prompt engineering, test-time compute, …

```
def react_agent(task_goal, db, env):
   plan = make_plan(task_goal)

   while (!env.done)
      obs = env.getobs()
      r = reason_for_action(task_goal, plan, obs)
      action = choose_action(task_goal, plan, obs, r)
      env.step(action)
```

And the agent loop continues until we reach an outcome!

Our ReAct agents use in-context examples

No prompt engineering, test-time compute, …

```
def react_agent(task_goal, db, env):
  plan = make_plan(task_goal, retrieve_for_planning(task_goal, db) )

   while (!env.done)
      obs = env.getobs()
      r = reason_for_action(task_goal, plan, obs, retrieve_for_reasoning(task_goal, db) )
      action = choose_action(task_goal, plan, obs, r, retrieve_for_acting(db, r) )
      env.step(action)
```

This is my entire agent, and this is all it does. Note that there is no complex hand-engineered prompting, test-time compute, etc. It's called a ReAct agent because it's Reasoning, then Acting. And since we have all these examples, we can go grab examples from the database at every step. Like any RAG system, we want the most relevant examples at any state.

So what data do we put in the database?

- We have an agent start solving tasks, and **if the task is solved, the DB grows by one—that's all we're gonna do**! Ran this on a few benchmarks:

(Read out this slide)

Benchmarks: ALFWorld

Embodied exploration and problem solving

TextWorld

Welcome!

You are in the middle of the room. Looking around you, you see a diningtable, a stove, a microwave, and a cabinet.

Your task is to:
Put a pan on the diningtable.

> goto the cabinet

You arrive at the cabinet. The cabinet is closed.

> open the cabinet

The cabinet is empty.

Embodied

> goto the stove

You arrive at the stove. Near the stove, you see a pan, a pot, a bread loaf, a lettuce, and a winebottle.

> take the pan from the stove

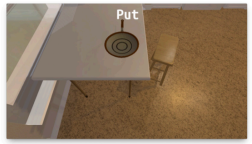
You take the pan from the stove.

> goto the diningtable

You arrive at the diningtable.

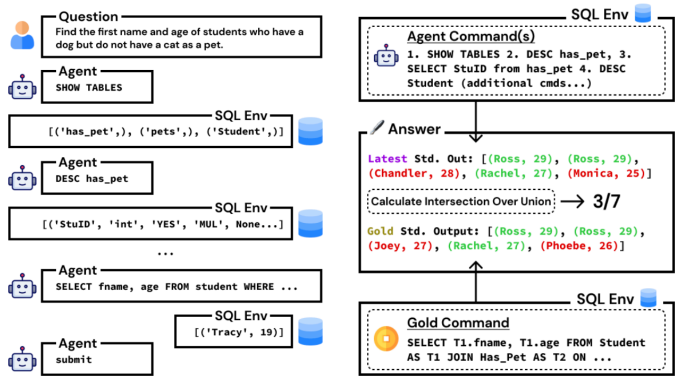
> put the pan on the diningtable

You put the pan on the diningtable.



Benchmarks: IC-SQL

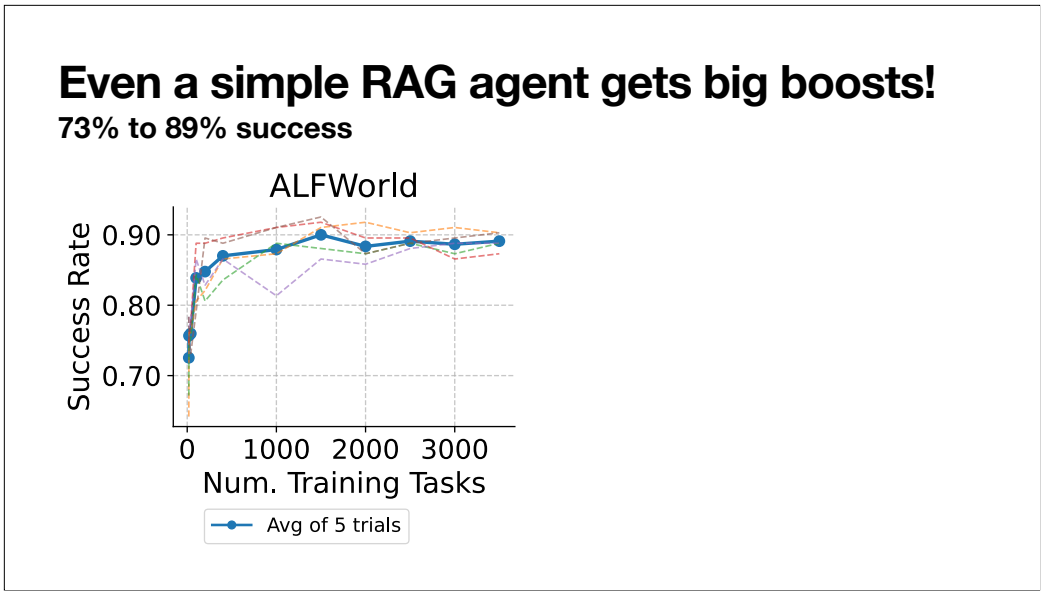
Multi-step database understanding for question-answering



Benchmarks: Wordcraft

Multi-step compositional reasoning

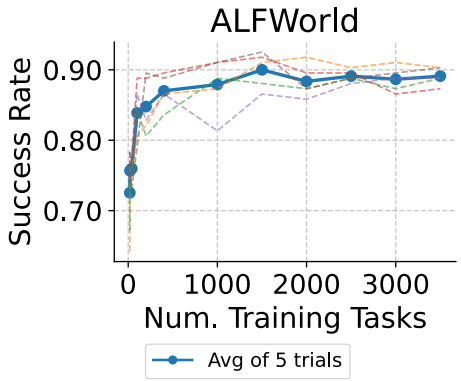
```
                                                                    t=0
  env> Goal:     create cyborg                                      r=0.0
       table:    [0:human, 1:metal, 2:fire, 3:life]
       selected: []
agent> 1 (metal)
                                                                    t=1
  env> table:    [0:human, 1:metal, 2:fire, 3:life]                 r=0.0
       selected: [metal]
agent> 3 (life)
                                                                    t=2
  env> table:    [0:human, 1:metal, 2:fire, 3:life, 4:robot]        r=1.0
       selected: []
agent> 0 (human)
                                                                    t=3
  env> table:    [0:human, 1:metal, 2:fire, 3:life, 4:robot]        r=0.0
       selected: [human]
agent> 4 (robot)
                                                                    t=4
  env> table:    [0:human, 1:metal, 2:fire, 3:life, 4:robot,        r=1.0
                  6:cyborg]
       selected: []
       Done.
```



On the Alfworld benchmark, even this naive RAG agent gets big boosts in performance! As the number of training tasks attempted on the x axis increases, the database gets bigger, and the relevance of retrieved examples improves—boosting agent success rates from 73% to 89%.

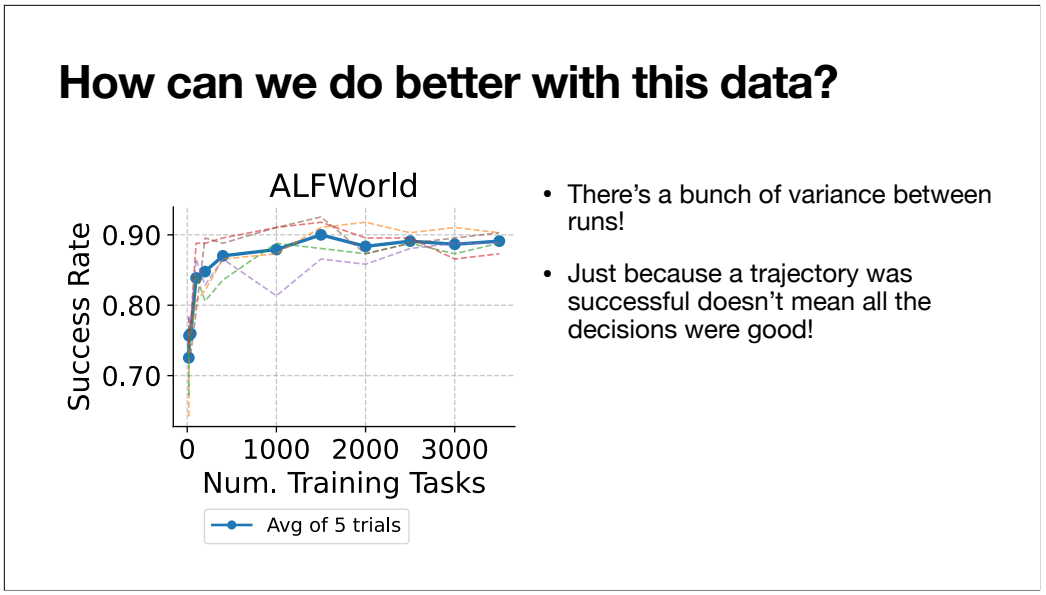
Even a simple RAG agent gets big boosts!

Agent performance in context…

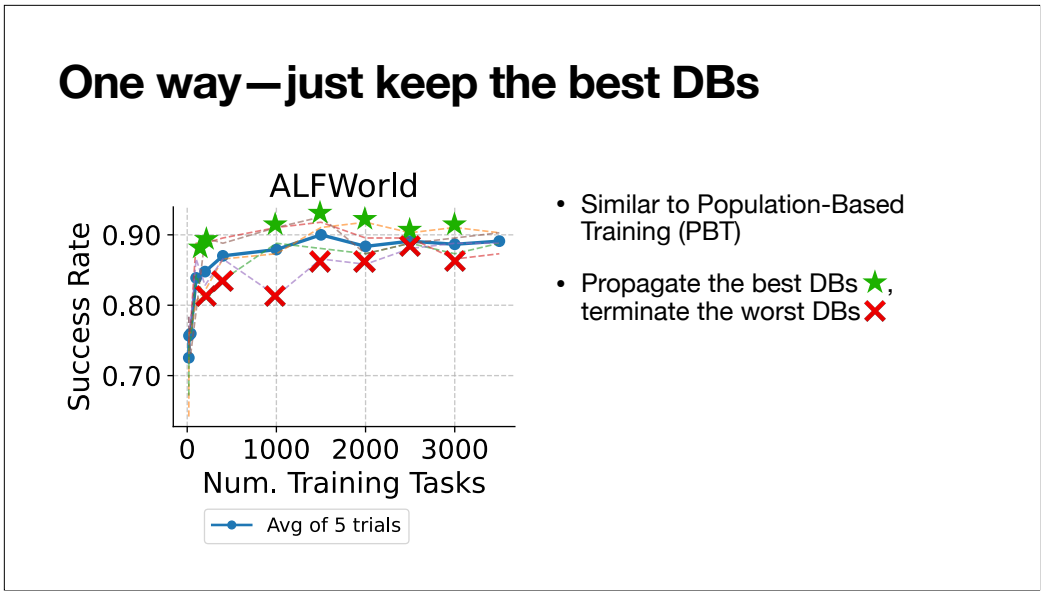


	Performance Boost (Δ)
Naive self-generated RAG DB	**+16**
Test-time scaling (pass@4)	+21
Model Improvement (4o-mini to 4o)	+15
Task-Specific Engineering	+18

This is basically the boost you get from using a better model (moving from GPT 4o-mini to 4o), or the 18 point boost from recent work applying a lot of task-specific engineering. It doesn't yet meet the boost from giving the agent 4 independent attempts per task, which is test-time scaling accompanied by a perfect verifier.



How can we do better with this data? Across the 5 trials on this graph, there's a lot of variance—so some collected databases are better than others! And just because a trajectory was successful doesn't mean all the decisions were good!



One way to do better is to just keep the best databases—and run more trials from high-performing collections of data! We carry out N training runnings, creating a population of databases. At certain checkopints, we stop and check which database leads to the best agent results on the training tasks so far… discard the lowest performing DB and replace it with a copy of the best performing.... and then continue training of the new population of N agents from there.

Another way—identify the most effective examples

Not all successful examples are made equal

Trajectory T1:
Provided in-context 7 times
Led to task success 1 time

Trajectory T2:
Provided in-context 7 times
Led to task success 6 time

And even within a “high-performing” database of examples, we really care about the examples that, when they are used, often lead to successful examples. Consider trajectory T1 in the DB retrieved by 7 novel trajectories that mostly fail. And a different trajectory T2 gets retrieved in 7 novel trajectories that mostly succeed. So T2 is a more effective in-context example than T1—and we’d prefer to keep T2 over T1.

Another way—identify the most effective examples

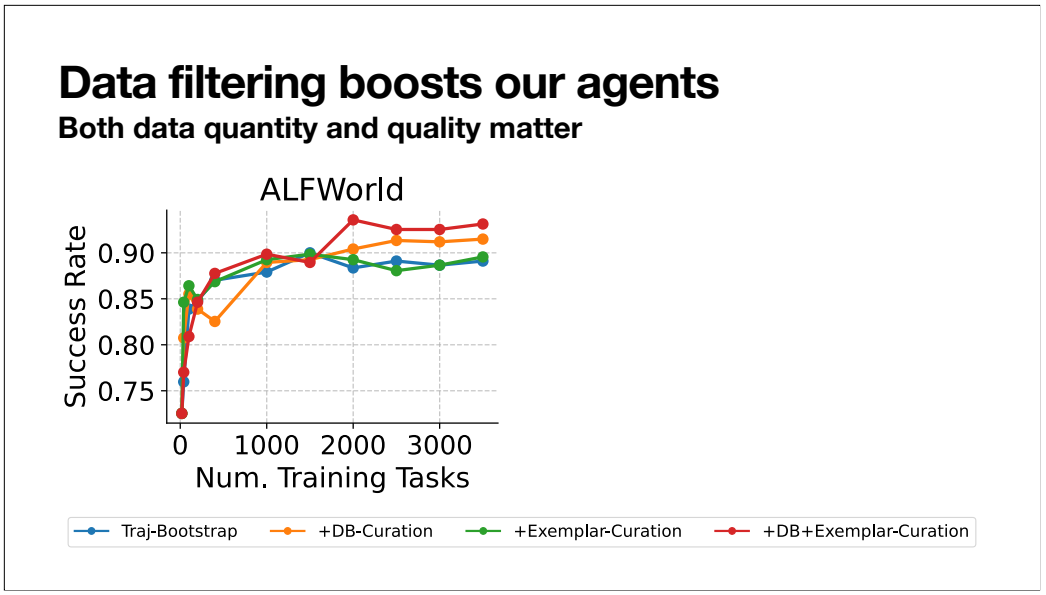
Keep the examples that are most effective when provided in-context

Trajectory T1:
Provided in-context 7 times
Led to task success 1 time

Trajectory T2:
Provided in-context 7 times
Led to task success 6 time

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

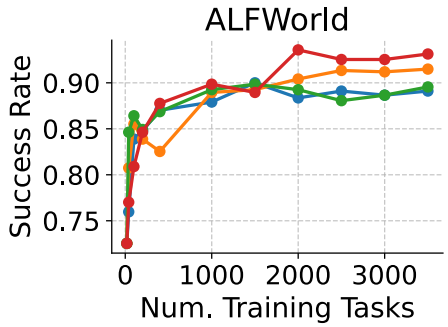
We apply this insight by computing the fraction of the time that each example leads to success when provided in-context to the agent, and for each task attempted we only keep the example which is most empirically effective across all databases.



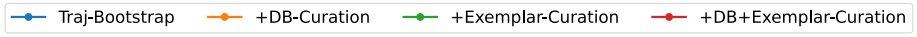
With a little bit of quality-based data filtering, our agent performs significantly better! This is the red line on the graph, versus the naive method in the blue line.

Our agents improve as they collect data

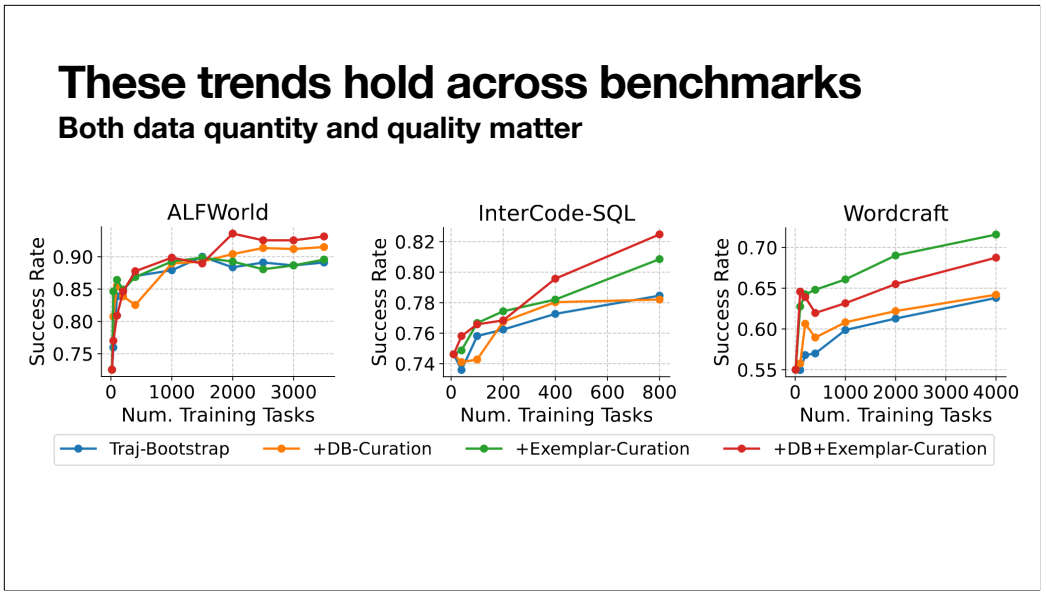
Both data quantity and quality matter



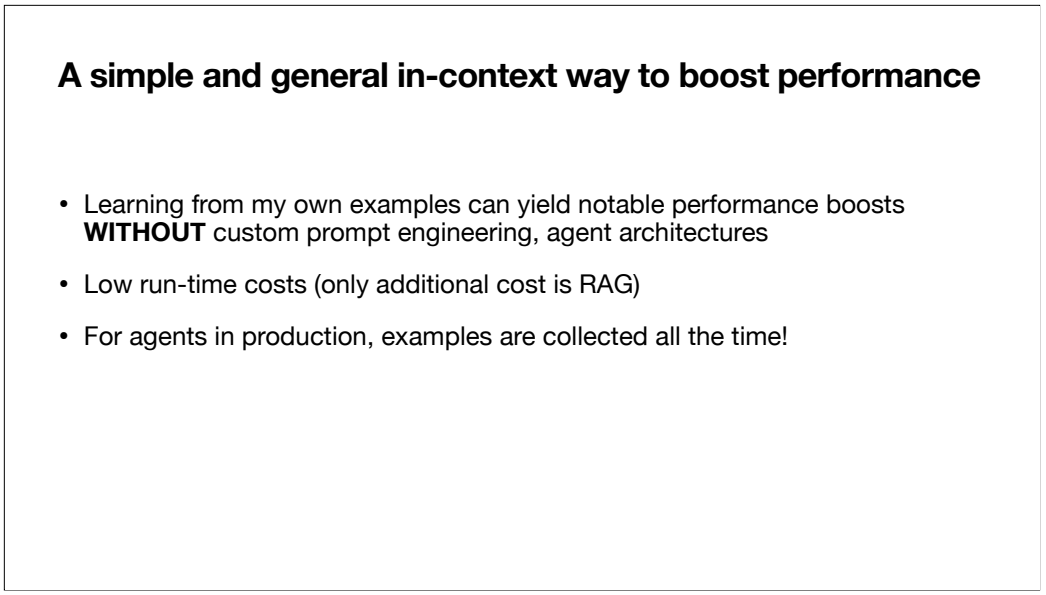
	Performance Boost (Δ)
Self-curated RAG DB	**+20**
Test-time scaling (pass@4)	+21
Model Improvement (4o-mini to 4o)	+15
Task-Specific Engineering	+18



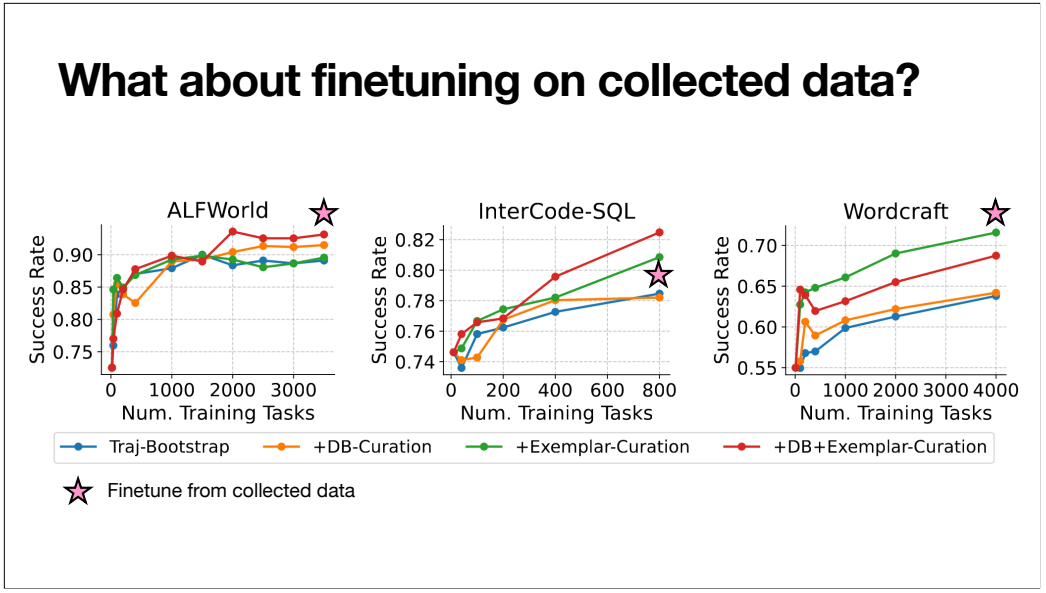
Putting this in context, we now have a 20-point boost over the baseline—basically matching the boost from having 4 attempts at each task via test-time compute and verification.



These trends hold across all the benchmarks we tested, and in particular on InterCode and Wordcraft our experiments clearly had not yet saturated in the data regime we tested.



So we've presented a simple and general in-context way to boost performance. Learning from the agents' own examples can yield notable performance boosts—without custom prompt engineering or agent architectures. This algorithm has low run-time costs—the only cost is from retrieval across a larger database. And in production use-cases, examples are collected all the time, so you can basically do this for free!

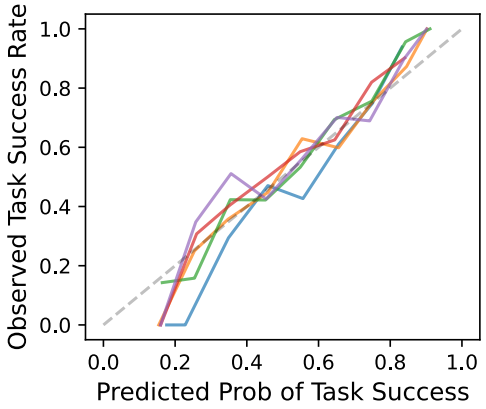


WHAT HAS REALLY HAPPENED here is that you have described a procedure for curating a dataset of helpful examples. And so those helpful examples ARE ALSO pretty helpful if they are used for model fine tuning instead of in context examples. The pink stars on the graphs show the performance of a fine-tuned agent on each of the three benchmarks—once we have thousands of examples we can slightly outperform the performance of our in-context agent on 2 out of the 3 benchmarks by fine-tuning on the collected data.

Another thing we can do with collected examples

Train a task success predictor for the agent!

- Anyone that's using agents: predicting success is critical
- Anyone that's serving LLMs: build routers to optimize inference



And by the way, if you have all this data, and you analyze it, you can build predictive models of your agents! This graph shows that for one of our benchmarks, we can use our collected data to predict the probability of task success—before the agent has even taken a single action. Predicted probability of success is on the x axis and observed success rates are on the y axis—and our predicted probability of task success closely tracks the actual task success of the agent—a perfect predictor would follow the dashed line. This is great whether you're looking to use agents yourselves, or even serving LLMs and looking to optimize inference costs with routers.

Can we do better? What if there's no verifier?

- We could certainly do better—via retrieval, data filtering, etc. We hope to see more research in this area.
- We've assumed that there is always a verifier—but that's certainly not true in practice. Can I have a human in the loop?