

RL without Tears:

Chapter 6: Agentic RL

Chenglong Wang
Northeastern University, China

WANGCHENGLONG@MAIL.NEU.EDU.CN

Hang Zhou
Northeastern University, China

STCEUM@GMAIL.COM

Tongran Liu
Institute of Psychology, Chinese Academy of Sciences, China

LIUTR@PSYCH.AC.CN

Jingbo Zhu
Northeastern University, China

ZHUJINGBO@MAIL.NEU.EDU.CN

Tong Xiao
Northeastern University, China

XIAOTONG@MAIL.NEU.EDU.CN

6 Agentic RL

Using RL to train agents, often referred to as *agentic RL*, is not a new concept. In classical machine learning, agentic RL typically involves training a domain-specific decision model from scratch. For example, in tasks such as playing complex games like Atari and Go (Mnih et al., 2013; Silver et al., 2017) or controlling robotic locomotion (Lee et al., 2024), the goal is to explore a structured environment and learn an optimal policy starting from random initialization.

With the emergence of LLMs as core components of autonomous systems, the role of RL has shifted from learning policies from scratch to adapting pretrained models for sequential decision-making in dynamic and open-ended environments. In this context, agentic RL aims to address a key limitation of LLMs: although they contain extensive world knowledge, their outputs are not inherently optimized for long-horizon interactions. For example, an SFT-trained LLM may successfully generate code for calling an external API, but it may fail to recover from unexpected execution errors during interaction.

In this section, we focus on the emerging paradigm of agentic RL for LLM-based agents. We first discuss how RL enables agents to acquire fundamental capabilities, including long-horizon planning and tool-integrated reasoning. We then introduce the role of environment design and scaling, which provide the essential foundation for training and evaluating agentic RL systems. We also discuss credit assignment, which provides more fine-grained feedback for agent learning. Finally, we explore how agents can continuously improve from their interaction experiences through mechanisms, including memory management, skill optimization, and trajectory refinement.

6.1 Building Agent Capabilities

LLM-based agents extend LLMs from passive response generators to interactive decision-making systems. Instead of producing a single response, agents continuously observe their environments, select actions, and update their behaviors through interaction. This interaction loop enables agents to solve complex tasks that require long-horizon execution. Although LLMs already possess strong capabilities, such as instruction understanding and reasoning, these capabilities are mainly developed for static text generation and may not be sufficient for dynamic interaction scenarios. When deployed as agents in real-world environments, LLMs need to further adapt their reasoning abilities to sequential decision-making, where they must decompose

high-level goals into executable steps, select appropriate actions, and recover from unexpected failures during execution.

Such agentic capabilities are difficult to acquire from static input-output demonstrations alone. The effectiveness of an individual decision often depends on its long-term consequences and the final task outcome. A locally reasonable action may eventually lead to failure after multiple interactions, while an unsuccessful attempt may provide valuable experience for future decisions. To address this challenge, RL has become a standard approach for training agentic systems, enabling agents to explore different behaviors and improve their abilities through environmental feedback (Singh et al., 2025c; Feng et al., 2025).

In this subsection, we focus on two key capabilities in agentic RL: *planning* and *tool use*. Planning enables agents to decompose complex goals into executable steps and make effective decisions over long horizons. Tool use allows agents to interact with external resources, such as knowledge sources and computational tools, thereby extending their capabilities beyond the information encoded in model parameters.

6.1.1 Planning

An agent’s autonomy refers to its ability to independently determine the steps required to complete a given task, even when these steps are not explicitly specified in the input. This capability requires the agent to infer the necessary procedure from the task description, decompose the goal into executable subgoals, and decide when and how to invoke external tools. Such autonomy is often reflected in the generation of a structured plan that guides the execution trajectory. For example, given a specific task description, the agent may produce an actionable plan with tool-use decisions as follows:

Input	<i>Environment:</i> Calculator, Database API <i>Task:</i> Weng earns \$12 per hour for babysitting. Yesterday, she babysat for 50 minutes. How much did she earn? Please save the result into the database.
Output	1. Extract the relevant numerical values and identify the required computation. 2. Call the <code>calculator</code> tool to evaluate the expression $12 \times (50/60)$. 3. Format the computed result into a valid JSON schema. 4. Invoke the <code>update_record</code> API to save the final earnings into the database. Next, I will execute the plan step by step.

Unlike single-turn response generation, agent planning requires a sequence of interdependent decisions, where early choices can affect subsequent tool use, environmental feedback, and final task success. Improving planning capability therefore requires not only teaching the model to generate reasonable intermediate steps, but also enabling it to adjust its behavior based on execution outcomes. To this end, researchers have explored various approaches to achieving this goal. Among them, one promising approach is RL, as it naturally formulates agent planning as sequential decision-making driven by environmental feedback and trajectory-level rewards. In practice, SFT provides a cold start by allowing the model to imitate high-quality planning trajectories, while RL further refines the agent through interaction-based feedback and trajectory-level optimization.

6.1.1.1 Supervised Planning Data

Before applying RL, SFT is commonly adopted to provide a cold start for LLM-based agents by teaching them basic planning and interaction behaviors (Chen et al., 2023; Zhang et al., 2024; Zeng et al., 2024). As illustrated in Figure 1, constructing SFT data for agent planning typically involves three stages: (1) preparing diverse environments and planning tasks; (2) synthesizing expert-level trajectories, consisting of action-observation sequences, through agent-environment interactions; and (3) selecting high-quality trajectories based on predefined evaluation criteria. Specifically, expert trajectories can be generated

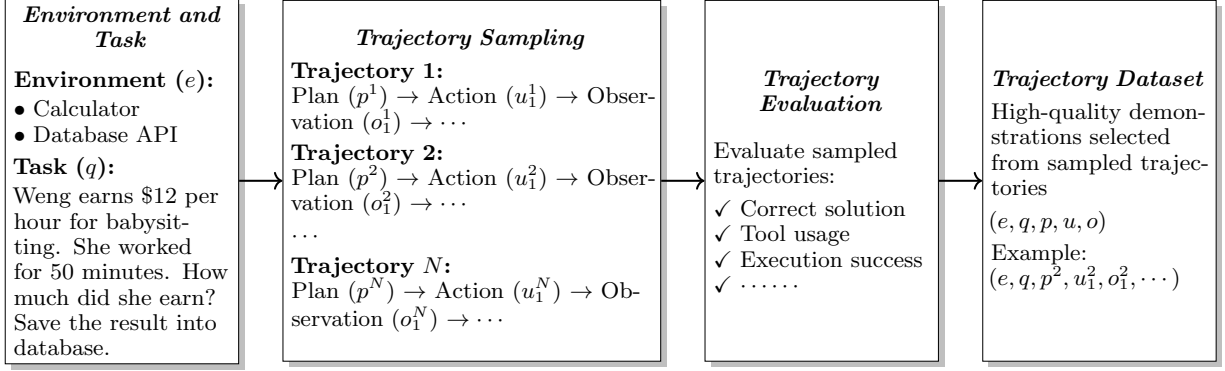


Figure 1: Overview of SFT data construction for improving agent planning.

by leveraging state-of-the-art LLMs as expert agents and retaining reliable demonstrations according to reward signals or task success criteria. Through this process, LLMs can learn planning behaviors from demonstrations and improve their ability to generate structured execution plans.

$$\mathbf{x} = [e, q] \quad (1)$$

$$\mathbf{y} = [p, u_1, o_1, \dots, u_T, o_T] \quad (2)$$

where e denotes the environment information, including available tools and external resources, and q represents the task description provided by users. p denotes the generated plan that decomposes the task into intermediate steps. u_t and o_t represent the agent’s behavior and the corresponding observation from the environment at the t -th interaction step, respectively. Specifically, u_t denotes the executable behavior performed by the agent, such as tool invocation, while o_t denotes the feedback returned by the environment after executing u_t . Therefore, $\{u_1, o_1, \dots, u_T, o_T\}$ forms a sequential interaction trajectory between the agent and the environment. It is worth noting that although u_t and a_t in Section ?? both represent “action”, they are defined at different levels of abstraction. Specifically, u_t represents a high-level interaction step in agent planning, such as calling a specific tool, whereas a_t represents the low-level token-level action in the standard LLM-based RL formulation, corresponding to the generation of an individual token by the language model.

Similar to LLM instruction tuning (Longpre et al., 2023; Zhou et al., 2023), the scale and quality of trajectory data significantly influence the effectiveness of agent training. High-quality trajectories provide explicit supervision signals for agents to learn how to decompose complex tasks, select appropriate tools, and interact with dynamic environments. Therefore, recent studies have explored constructing trajectory-based instruction tuning datasets to enhance the planning capabilities of LLM-based agents.

A straightforward approach is to utilize generated trajectories for instruction tuning. However, trajectories collected from LLM-based agents may contain unsuccessful plans, ineffective tool usage, or execution failures, which can introduce noisy supervision. Therefore, existing methods typically employ filtering strategies based on task success criteria or heuristic rules to select high-quality demonstrations. For example, AgentTuning constructs AgentInstruct by collecting interaction trajectories and filtering unsuccessful samples, while FireAct investigates the impact of diverse trajectory data from multiple tasks and reasoning paradigms for improving agent fine-tuning (Zeng et al., 2024; Chen et al., 2023).

Beyond selecting high-quality trajectories, another challenge lies in effectively representing the complex interaction processes within trajectories. Unlike conventional instruction tuning, agent trajectories contain not only final responses but also intermediate planning processes and multi-step interactions with environments. Therefore, how to organize and annotate different components of trajectories becomes crucial for enabling agents to acquire planning and execution capabilities. To address this challenge, AgentLUMOS introduces

a modular training framework that explicitly separates high-level planning and low-level execution (Yin et al., 2024). Specifically, its planning module learns to generate high-level subgoals, while its grounding module learns to translate these subgoals into executable actions through external tools. Such a modular design enables agents to acquire different capabilities from structured trajectory annotations.

Furthermore, even with effective trajectory selection and representation, scaling the generation of diverse and high-quality trajectory data remains challenging. Existing approaches often rely on manually designed environments and planning tasks, which limits the diversity and coverage of collected trajectories. Designing new environments requires substantial human expertise, while constructing tasks with appropriate difficulty levels remains difficult. To address this limitation, AGENTGEN explores automatically generating diverse environments and planning tasks with LLMs, enabling large-scale synthesis of trajectory data with varying task complexity for agent training (Hu et al., 2025).

6.1.1.2 Learning to Plan with RL

Although SFT can provide a useful cold start for agent planning, it mainly teaches the model to imitate offline demonstrations. This limits its ability to improve beyond the quality and coverage of the collected trajectories. In contrast, RL further optimizes planning through direct interaction with the environment, where the agent receives feedback based on the outcome of its generated plans and executed behaviors.

Under the agent planning formulation, the LLM-based agent can be viewed as a high-level policy that generates a plan and then produces a sequence of executable behaviors (Li et al., 2025). Given the environment information e and task description q , the agent first generates a plan p and then interacts with the environment through a sequence of behaviors and observations:

$$\tau = [p, u_1, o_1, \dots, u_T, o_T] \quad (3)$$

The goal of RL is to optimize the agent policy so that the generated trajectory receives higher feedback from the environment. This objective can be written as

$$\theta^* = \arg \max_{\theta} \mathbb{E}_{\tau \sim \pi_{\theta}(\cdot|e,q)} [R(\tau)] \quad (4)$$

where π_{θ} denotes the agent policy parameterized by the LLM, and $R(\tau)$ denotes the trajectory-level reward that evaluates the overall quality of the planning process. In practice, this reward can be defined according to task success. The objective can then be optimized using standard RL algorithms introduced in Section ??, such as PPO or GRPO.

Compared with SFT, RL provides two important advantages for agent planning. First, it enables the agent to learn from execution outcomes rather than only from static demonstrations. If a plan leads to failed tool execution or poor environmental feedback, the agent can be penalized and encouraged to explore alternative behaviors. Second, RL supports trajectory-level credit assignment, allowing the model to adjust earlier planning decisions according to later outcomes. This is crucial for agent planning, where early subgoal decomposition or tool selection can substantially affect the final task success.

6.1.2 Tool Use

Tool use is another fundamental capability of LLM-based agents. Tools extend the agent beyond its internal parameters. They allow the agent to access external knowledge and execute actions in external systems. Early studies show that LLMs can use tools through prompt-based reasoning-action patterns (Chang et al., 2024; Yao et al., 2023). ReAct is a representative example, where the model alternates between reasoning steps and tool-use actions (Yao et al., 2023). However, prompt-based tool use is often unstable. It depends heavily on the capability of the model. Thus, SFT methods train LLMs on tool-use trajectories, so that the model can learn when and how to invoke tools from demonstrations (Komeili et al., 2022; Schick et al., 2023). Still, SFT mainly teaches imitation. It cannot directly optimize whether the agent should use a

tool, which tool it should select, or how to balance tool benefit with tool cost. RL provides a natural framework for efficient tool use. We can define rewards based on task success, tool-use correctness, or tool-use completeness (Qian et al., 2025; Singh et al., 2025b).

6.1.2.1 Prompt-based Tool Use

Prompt-based tool use is a simple approach to enable LLM-based agents to interact with external tools without updating model parameters. However, the model may not naturally know when to invoke a tool or how to organize tool-use behaviors. Thus, we usually need to provide tool descriptions and a few demonstrations in the prompt. These demonstrations show the model how to alternate between reasoning, tool invocation, and observation. At each step, the model first generates a reasoning trace, then produces a tool-use action. After the tool returns an observation, the model uses this feedback to update its next decision. For example,

Tools	calculator(expression): return the numerical result of an expression. update_record(json): save a JSON record into the database.
Demonstration	Task: Tom earns \$15 per hour. He worked for 2 hours. How much did he earn? Please save the result into the database. Thought: I need to compute the total payment. Action: <code>calculator(15 × 2)</code> Observation: 30 Thought: The payment is 30 dollars. I need to save it into the database. Action: <code>update_record({"earning": 30})</code> Observation: success Answer: Tom earned \$30, and the result has been saved into the database.
Current Task	Weng earns \$12 per hour for babysitting. Yesterday, she babysat for 50 minutes. How much did she earn? Please save the result into the database.
Output	Thought: I need to compute the payment based on the hourly wage and working time. Action: <code>calculator(12 × 50/60)</code> Observation: 10 Thought: The payment is 10 dollars. I need to save it into the database. Action: <code>update_record({"earning": 10})</code> Observation: success Answer: Weng earned \$10, and the result has been saved into the database.

This demonstration teaches the model a tool-use pattern. The model learns to reason about the task, invoke an external tool when necessary, read the returned observation¹, and continue the execution process.

6.1.2.2 Supervised Tool-use Data

To make tool use more reliable, we can further train LLMs on tool-use trajectories. In this setting, each training example contains a user task, available tool descriptions, and a demonstrated tool-use process. The model learns when to invoke a tool, which tool to select, how to construct valid arguments, and how to use the returned observation. For example, we can construct a training example as follows:

$\mathbf{x}$ = Task: Weng earns \$12 ... Tools: calculator(expression), ... , update_record(json)
 $\mathbf{y}$ = Thought: compute the payment; Action: calculator(12 × 50/60); ...; Answer: Weng earned \$10.

¹We refer to the response returned by an external tool as an “observation”.

With such data, SFT teaches the model to imitate the demonstrated tool-use pattern. The model can learn the format of tool invocation and the role of observations in later generation.

A common way to construct such data is to manually annotate tool-use trajectories. However, this requires annotators to know which tool should be used, how to write valid arguments, and how to judge whether the returned observation is useful. This process is costly and difficult to scale to many tools and tasks.

A representative approach to addressing this data construction problem is Toolformer, which uses a self-supervised method to generate tool-use data (Schick et al., 2023). Instead of relying on large-scale human annotations, Toolformer starts from only a few demonstrations for each API. It then lets the language model annotate a large text corpus with possible API calls. After executing these calls, Toolformer keeps only the API calls that help the model better predict future tokens. The filtered data is then used to fine-tune the model. Through this process, the model learns when to call a tool, what arguments to pass, and how to incorporate the tool result into later generation.

Beyond learning to use a small set of tools, later studies further scale tool-use training to larger and more realistic API spaces. API-Bank builds a benchmark and training set for tool-augmented LLMs, where models need to plan, retrieve, and call APIs in executable environments (Li et al., 2023). ToolLLM constructs ToolBench with more than 16,000 real-world APIs and uses automatically generated instruction-solution trajectories to train ToolLLaMA (Qin et al., 2024). Gorilla focuses on API calling at scale and fine-tunes LLaMA-based models to generate accurate API calls. It also uses a document retriever to reduce API hallucination and adapt to changing API documents (Patil et al., 2024).

6.1.2.3 Learning to Use Tools with RL

Although SFT can teach LLMs to imitate tool-use demonstrations, it is still limited by the coverage and quality of offline trajectories. The model mainly learns the tool-use patterns that appear in the supervised data. As a result, it may fail to generalize to unfamiliar tools, complex tool combinations, or new interaction patterns. More importantly, SFT does not directly optimize whether a tool call is necessary, whether the selected tool is appropriate, or whether the returned observation improves the final answer. This limitation becomes more serious in multi-step tool-use scenarios, *where the model needs to decide when to call a tool, how to formulate the tool input, how to interpret the tool output, and when to stop using tools*.

To address this limitation, researchers have explored RL for enhancing tool use. Instead of only imitating fixed demonstrations, the model can interact with tools during rollout and receive feedback from task outcomes or tool execution results. Given a user task q and a set of available tools $\mathcal{T}$, the model generates a tool-use trajectory:

$$\tau = [h_1, u_1, o_1, \dots, h_T, u_T, o_T, y] \quad (5)$$

where h_t denotes the reasoning state at the t -th step, u_t denotes the high-level tool-use action, o_t denotes the observation returned by the tool or environment, and y denotes the final answer. Here, h_t can include natural language reasoning, while u_t represents an executable behavior such as selecting a tool and providing arguments. The goal of RL is to optimize the policy so that the generated trajectory receives a higher reward:

$$\max_{\theta} \mathbb{E}_{\tau \sim \pi_{\theta}(\cdot|q, \mathcal{T})} [R(\tau)] \quad (6)$$

In practice, this objective can be optimized with standard RL algorithms such as PPO or GRPO. Recent studies show that such outcome-driven optimization can help LLMs learn more adaptive tool-use behaviors, such as deciding when to search, when to execute code, and how to revise a tool call after receiving an error message (Jin et al., 2025; Chen et al., 2025; Feng et al., 2025; Singh et al., 2025a).

However, RL-based tool use also introduces several unique challenges. First, the action space is more complex than ordinary text generation. A tool-use action may involve tool selection, argument generation,

execution timing, and response integration. A small error in any part can lead to tool failure. Second, the reward signal is often sparse and delayed. The environment may only provide a final success signal after the whole trajectory is completed, making it difficult to identify which tool call causes success or failure. Third, tool use may introduce additional costs. Unnecessary tool calls can increase latency and computation cost, while insufficient tool use may lead to incorrect answers. Therefore, the model needs to learn not only how to use tools, but also how to use them efficiently.

A key direction is to design more informative rewards for tool use. Instead of using only a final-answer reward, we can decompose the reward into multiple components:

$$R(\tau) = R_{\text{ans}}(\tau) + \alpha R_{\text{fmt}}(\tau) + \beta R_{\text{exec}}(\tau) + \gamma R_{\text{tool}}(\tau) - \lambda C_{\text{tool}}(\tau) \quad (7)$$

where R_{ans} measures final answer correctness, R_{fmt} measures whether the model follows the required tool-use format, R_{exec} measures whether tool calls can be successfully executed, and R_{tool} measures whether the model selects appropriate tools and passes valid arguments. C_{tool} denotes the cost of tool use, such as the number of tool calls, latency, or computational expense. The coefficients α , β , γ , and λ control the relative importance of different reward terms.

In application, this reward decomposition provides denser feedback than final-answer rewards. For example, if the model selects the correct tool but passes an invalid argument, the execution reward can penalize the invalid call while preserving credit for the correct tool selection. If the model reaches the correct answer but uses many unnecessary tool calls, the cost term can discourage inefficient behavior. ToolRL systematically studies this reward design problem and shows that fine-grained reward decomposition can make RL training more stable and effective for tool selection and tool application (Qian et al., 2025).

Another direction is to use RL to improve strategic tool use. For example, ReTool first builds a cold-start model with code-augmented reasoning traces and then applies RL with real-time code execution. During rollout, the model interleaves natural language reasoning with code execution, observes execution results, and revises its subsequent reasoning. This allows the model to discover when and how to invoke a code interpreter based on outcome feedback rather than human-written rules (Feng et al., 2025). Search-R1 and ReSearch follow a similar idea in retrieval-augmented reasoning. They train models to generate search queries during reasoning and use retrieved information to update later steps, rather than relying on fixed retrieval pipelines or hand-crafted search prompts (Jin et al., 2025; Chen et al., 2025).

So far, we have introduced how RL can enhance the core capabilities of LLM-based agents. Since LLM-based agents are built upon underlying LLMs, their RL training can naturally leverage the general LLM-based RL algorithms introduced in Section ?? . However, agentic RL introduces additional challenges beyond standard response optimization. A key challenge is *how to obtain high-quality feedback from environments* and *how to effectively use these signals to improve agent behaviors*. As a result, recent work on agentic RL requires not only appropriate RL algorithms, but also specialized approaches for addressing the unique challenges introduced by agentic settings (Huo et al., 2026; Liu et al., 2026).

6.2 Environment Design and Scaling

The previous subsections focus on optimizing the agent policy for planning and tool use. In this subsection, we turn to another important component of agentic RL: the environment that produces interaction experience. An environment is more than a testbed. It determines which actions the agent can perform, what observations it receives, how the underlying state changes after each action, and how task success is evaluated. More formally, we can represent an environment as

$$e = (\mathcal{S}, \mathcal{U}, \mathcal{O}, P_e, V_e) \quad (8)$$

where $\mathcal{S}$ is the state space, $\mathcal{U}$ is the set of available actions or tool invocations, and $\mathcal{O}$ is the observation space. The transition function P_e determines how an action changes the environment state, while the

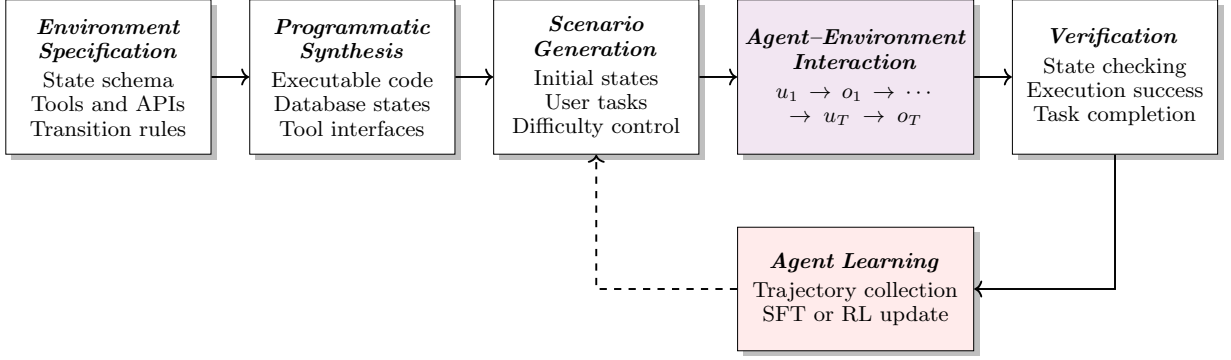


Figure 2: Overview of environment synthesis and interaction for agentic RL. Agentic RL enables continuous learning through an interaction-learning loop, where agents collect trajectories from synthesized environments and enhance their capabilities.

verifier V_e determines whether the resulting state satisfies the task objective. This formulation describes how the environment produces observations and feedback, rather than how the agent generates its actions.

The quality of an environment directly affects what an agent can learn. For example, an agent trained in a narrow environment may simply memorize specific APIs, task templates, or interaction patterns. An unreliable environment may return inconsistent observations or incorrect rewards, introducing noise into the learning process. Ideally, an effective environment should provide diverse tasks, executable interactions, reliable state transitions, and verifiable outcomes.

To encourage generalization, we can train the agent over a distribution of environments rather than within a single fixed environment:

$$\theta^* = \arg \max_{\theta} \mathbb{E}_{e \sim p(\mathcal{E}), q \sim p_e(\mathcal{Q})} \mathbb{E}_{\tau \sim \pi_{\theta}(\cdot | q, e)} [R_e(\tau)] \quad (9)$$

where $p(\mathcal{E})$ denotes the environment distribution, and $p_e(\mathcal{Q})$ denotes the task distribution within environment e . The interaction trajectory τ follows the definition introduced in the previous subsections. This objective shows that agent performance depends not only on policy optimization, but also on the coverage and quality of the environments used for training.

However, constructing high-quality environments manually is expensive and difficult to scale. Real-world systems may be inaccessible and costly. Purely language-based simulations are easier to build, but they may generate inconsistent state transitions (Li et al., 2026). Thus, recent studies explore procedural and programmatic environment synthesis, where executable programs and structured states are used to provide a scalable and reliable interaction experience.

This process is illustrated in Figure 2. It typically consists of the following steps:

- **Environment Specification.** We first define the state schema², available tools and APIs, and transition rules of the environment.
- **Programmatic Synthesis.** We then implement the environment as an executable system. This process typically converts the structured specification into program code, database tables, and callable tool interfaces.
- **Scenario Generation.** Based on the environment specification, we generate diverse initial states and user tasks. The tasks should cover different goals, constraints, and difficulty levels. For example,

²A state schema specifies the structured variables used to represent the current status of an environment, together with their types and possible values. For example, in a travel-booking environment, it may define fields for available flights, user preferences, and existing reservations.

in a shopping environment, a simple task may ask the agent to purchase a single item, while a more difficult task may require it to compare multiple products.

- **Agent–Environment Interaction.** The agent interacts with the generated environment through multiple rounds of actions and observations. At each step, the agent selects a tool and provides the required arguments. The environment executes the action and returns an observation.
- **Outcome Verification.** After the interaction is completed, a verifier checks whether the task has been successfully solved. The verifier may inspect tool execution results, database states, or other structured records.
- **Agent Learning.** Finally, the verified trajectories are used to improve the agent. On the one hand, successful trajectories can serve as demonstrations for SFT, while task-level or step-level feedback can support RL training. On the other hand, failed interactions are also valuable because they reveal weaknesses in the current policy and gaps in the task distribution. This can guide the generation of targeted scenarios that exercise the agent’s weak capabilities and provide more informative experience for subsequent training (Chen et al., 2026; Sun et al., 2026). More broadly, this process can be viewed as learning from agentic experience, which we will discuss in Section 6.4.

Despite this progress, several important questions remain. First, how can we scale both the number and diversity of environments? Second, how can we ensure that generated environments are executable and verifiable? Third, how can agents generalize across heterogeneous and previously unseen environments?

One direction focuses on scaling environment diversity. RandomWorld procedurally generates executable tools and compositional tasks for tool-use agents (Sullivan et al., 2025). Unlike static datasets that contain only predefined trajectories, these generated environments support online execution and allow agents to explore different tool combinations during training. AgentScaler further constructs heterogeneous simulated environments and separates the learning of general function-calling behaviors from domain-specific adaptation (Fang et al., 2026). At a larger scale, DeepSeek-V3.2 synthesizes diverse environments and complex tasks for agentic post-training, showing that broad interaction experience is important for long-tail generalization (DeepSeek-AI, 2025).

Another direction focuses on executability. EnvScaler first constructs environment skeletons that specify state schemas, available tools, and interaction rules. It then generates multiple task scenarios within each environment (Song et al., 2026). Agent World Model follows a similar idea by implementing synthetic environments with executable code and database-backed states (Wang et al., 2026c). Compared with purely language-based simulations, these programmatic environments produce more consistent state transitions and generate tool outputs through actual execution.

Verifiability is equally important because RL requires reliable feedback. Instead of evaluating only the textual final response, an environment can check whether the agent has produced the desired state change. Given the final environment state s_T^e and the task goal g , we can define a state-based reward as

$$R_e(\tau) = V_e(s_T^e, g) \quad (10)$$

where $V_e(\cdot)$ may be implemented using executable tests, database queries, or rule-based validators. For example, in a database operation task, the verifier can directly check whether the required record has been inserted correctly. This provides more reliable feedback than comparing the generated answer with a reference text. Both EnvScaler and Agent World Model use such mechanisms to connect agent actions with observable state changes.

Scaling environments also introduces substantial heterogeneity. Different environments may vary in task difficulty, available tools, and interaction length. These differences can make RL training unstable and cause the agent to overfit to frequently sampled or easier environments. AutoForge addresses this issue by synthesizing difficult but verifiable tasks and estimating learning signals at the environment level (Cai et al., 2025). This design reduces the influence of unstable simulated interactions and improves training across heterogeneous environments.

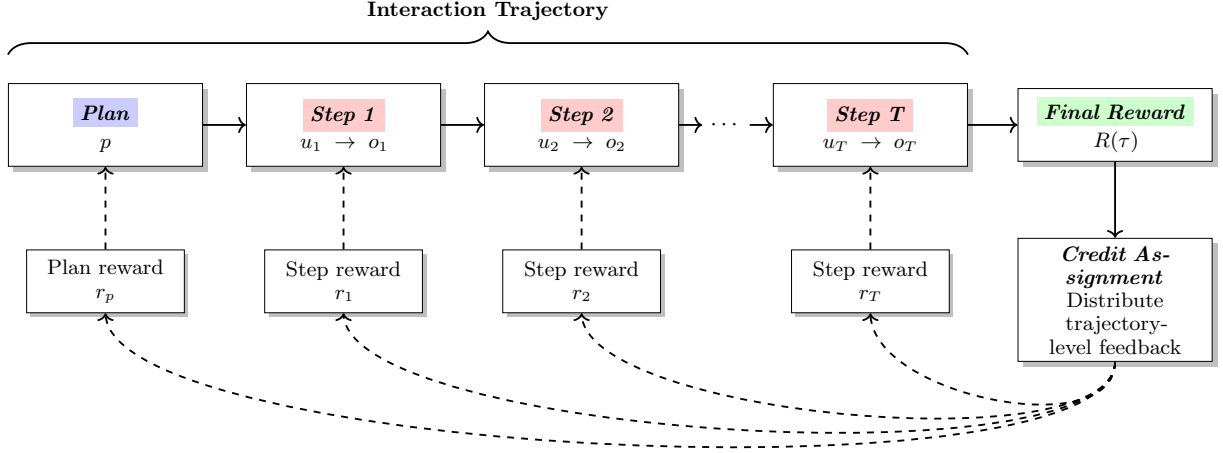


Figure 3: Illustration of credit assignment in agentic RL.

6.3 Credit Assignment

Credit assignment is not unique to LLM-based agents. It is a long-standing challenge in traditional RL (Sutton, 1984; Zhou et al., 2020). When rewards are delayed, the agent must decide which past actions should be reinforced or suppressed. Classical methods, such as temporal-difference learning and eligibility traces, address this issue by propagating reward information backward along the trajectory. In agentic RL, credit assignment becomes more challenging because each interaction step may involve high-level planning, tool invocation, and environmental feedback, rather than a single low-level action. Therefore, credit assignment often needs to operate at both the trajectory level and the step level.

This challenge is especially important for agent planning. In many agent tasks, the environment only provides a sparse reward after the entire trajectory is completed. However, the final success or failure may come from an earlier planning decision, an incorrect tool invocation, or an ineffective response to an intermediate observation. If we directly assign the same trajectory-level reward to all interaction steps, the supervision can become noisy. The agent may fail to identify which decisions should be reinforced and which should be suppressed.

Formally, given a trajectory $\tau = [p, u_1, o_1, \dots, u_T, o_T]$, a simple RL objective uses a trajectory-level reward $R(\tau)$ to optimize the whole planning process. However, such a reward only provides coarse feedback on the overall outcome. Therefore, credit assignment aims to decompose the trajectory-level reward into step-level signals:

$$R(\tau) \rightarrow \{r_p, r_1, \dots, r_T\} \quad (11)$$

where r_p evaluates the quality of the generated plan, and r_t evaluates the contribution of the t -th interaction step (u_t, o_t) to the final task outcome.

The mechanism of credit assignment is illustrated in Figure 3. This decomposition enables the agent to distinguish whether failure comes from an unreasonable plan, an invalid tool call, or a poor adaptation to environmental feedback (Li et al., 2025; Xi et al., 2026; Wang et al., 2026a). For example, consider the task of computing the babysitting payment and saving the result into a database. Suppose the agent generates a reasonable plan and correctly calls the calculator, but fails to invoke the database API with the required JSON format. If we only use a final trajectory-level reward, the whole trajectory may receive a low reward, e.g., $R(\tau) = 0.3$, even though the early planning and calculation steps are correct.

For illustration, we can define rule-based step-level rewards using predefined verification criteria, such as whether the plan includes all necessary steps, whether the calculation is correct, whether the JSON schema is valid, and whether the database update succeeds. These step-level rewards are diagnostic signals rather

than a conservative numerical decomposition, so they need not sum to the trajectory-level reward. Then, we can use credit assignment to decompose the final trajectory-level feedback into step-level rewards:

$$R(\tau) = 0.3 \quad \Rightarrow \quad \{r_p = 0.4, r_1 = 1.0, r_2 = 1.0, r_3 = 0.7, r_4 = 0.0\} \quad (12)$$

where r_p indicates that the overall plan is reasonable, r_1 and r_2 indicate that the agent correctly extracts the numerical values and performs the calculation, r_3 indicates that the JSON formatting is partially correct, and r_4 indicates that the database update fails. In this way, the agent receives more precise feedback: it should preserve the correct planning and calculation steps while improving the database update step.

6.4 Learning from Agentic Experience

AI systems are gradually moving from an era dominated by human-generated data toward an era in which agents increasingly learn from their own experience (Silver & Sutton, 2025). As discussed in Section 6.1, supervised data remains useful for teaching agents basic behaviors, such as planning, tool invocation, and response formatting. However, human demonstrations alone are unlikely to cover the full range of situations that an agent may encounter in complex and dynamic environments. One promising direction is to enable agents to learn from their own interaction experience. Through continuous interaction with environments, agents can collect successful and failed trajectories, and improve their future behaviors via these trajectories. In recent literature, this process is often referred to as *agent self-evolution*, where agents continuously enhance their capabilities by leveraging accumulated experience. Here we discuss three commonly used approaches for learning from agentic experience.

The first approach is **memory management** through agentic experience. In practice, an agent can store useful information from previous interactions. When facing a new task, the agent can retrieve relevant memories and use them as additional context to guide its decisions. From the perspective of in-context learning, this process can also be viewed as a form of *learning*, where the agent improves its behavior by incorporating previous experience into the current context. In this way, one example of learning from experience is to update the memory based on interaction outcomes. For example, given a task, the agent can store both successful and failed trajectories in its memory system. When solving similar tasks in the future, the agent can retrieve these experiences and avoid previous mistakes, thereby improving task performance. This approach is usually simple and does not require additional training of the agent.

The second approach is **skill optimization**. One challenge of using memory to learn from experience is scalability. As agent experience continuously accumulates, the memory size will keep growing. Moreover, a large amount of noisy and redundant information may also be introduced into the memory. As a result, efficiently storing and retrieving relevant information becomes increasingly difficult. Instead of storing individual experiences in memory, skill aims to identify common patterns across experiences and transform them into abstract behaviors. These skills provide a more compact representation of experience and allow the agent to transfer experience across different tasks. Similar to memory optimization, we can also update and refine the skills of agents to achieve learning from experience.

The third approach is to learn from experience via **trajectory refinement**. From this perspective, test-time scaling in agents can also be viewed as a form of learning from experience. Instead of updating the model parameters, the agent improves its current solution by leveraging immediate feedback during task execution. For example, the agent can use tool execution results or environment feedback to identify mistakes and refine its trajectory. Through repeated attempts and continuous refinement, the agent can gradually improve its decision-making process and achieve better task performance.

The three approaches mentioned above can be implemented through various techniques. Since this paper focuses on RL, we will introduce RL-based methods for these approaches in detail in the following sections. It is worth noting that, although these approaches do not always rely on traditional parametric updates, they can still be formulated under the RL paradigm described in Eqs. (3) and (4), where agents improve their behaviors via feedback from experience.

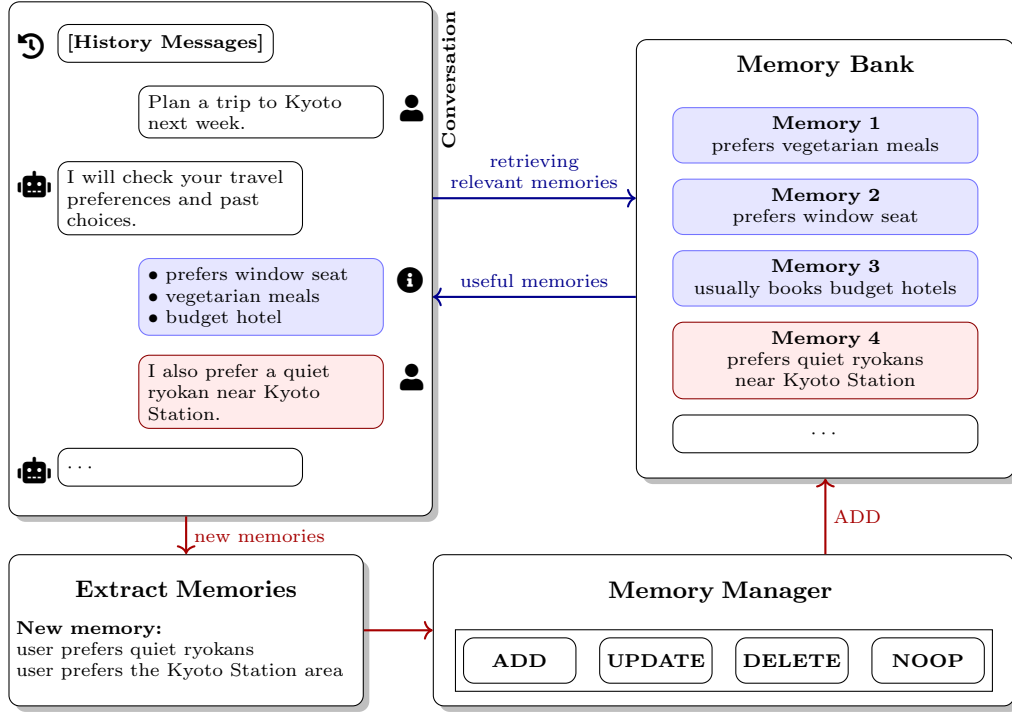


Figure 4: An overview of a memory system for agentic interaction. We take a travel-planning scenario as an example, where the user asks the agent to plan a trip to Kyoto. The agent first retrieves relevant user preferences from the memory bank, such as vegetarian meals, window seats, and budget hotels, and uses them to support the current conversation. During the interaction, the user provides new information, such as a preference for quiet ryokans near Kyoto Station. This new information is extracted as memory and passed to a memory manager, which updates the memory bank by selecting an operation such as ADD. Blue highlights denote retrieved memories, while red highlights denote newly extracted memories and their update path back to the memory bank.

6.4.1 Memory Management

Memory management is a direct way for agents to learn from agentic experience. During interaction with an environment, an agent may observe useful information. If this information is discarded after the current task, the agent must solve similar problems from scratch in the future. Therefore, the goal of agent memory is to store useful information from previous interactions and retrieve it when needed. As illustrated in Figure 4, a typical memory system usually consists of three main stages (Chhikara et al., 2025):

- **Memory Retrieval.** The agent retrieves relevant memories from the memory bank to support the current interaction. The retrieved information can take various forms depending on how the memory bank is constructed. For example, the memory bank may contain successful trajectories from similar tasks, which can provide reusable solutions for the current decision. It may also store user-specific preferences, such as frequently selected hotels or preferred travel styles.
- **Memory Extraction.** After the agent completes the current task based on retrieved information, useful information can be extracted from the interaction process and stored for future use. For example, if a user says that they are vegetarian, we can extract a memory such as “the user prefers vegetarian food”. This process can be performed by the agent itself or by other agents.
- **Memory Update.** The agent updates the memory bank by integrating the newly extracted information with existing memories. A straightforward approach is to store all extracted information. However, this strategy is impractical because the amount of stored information continuously grows

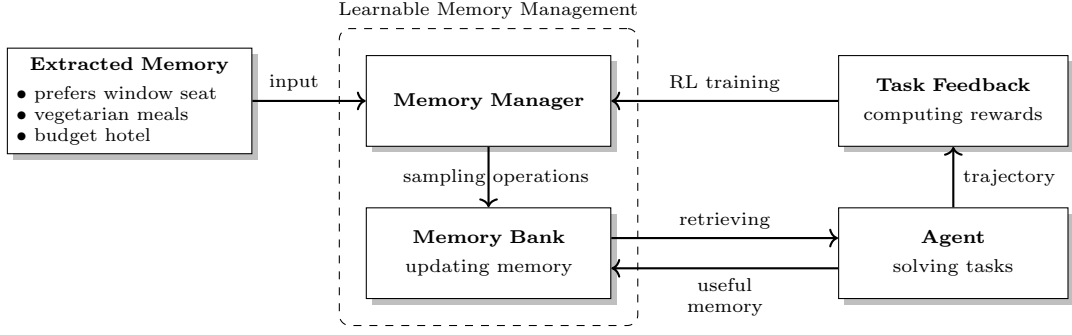


Figure 5: Overview of training a memory manager with RL.

as the agent operates over time, making memory retrieval increasingly inefficient and introducing a large amount of redundant information. In contrast, a more advanced approach is to selectively update the memory. Specifically, the memory system can formulate memory maintenance as an operation selection problem. Given newly extracted information, a *memory manager* selects one of several operations: $\text{op} \in \mathcal{A}^m = \{\text{ADD}, \text{UPDATE}, \text{DELETE}, \text{NOOP}\}$, where **ADD** creates a new memory entry, **UPDATE** modifies an existing memory with newly observed information, **DELETE** removes outdated or contradictory memories, and **NOOP** keeps the memory bank unchanged. We can typically implement this memory manager by prompting an LLM to select appropriate operations based on the current interaction and existing memories.

In learning from experience, it is easy to observe that the performance of memory-based learning heavily depends on the accuracy of the memory manager. Although prompting an LLM can enable it to select memory operations, such memory systems still rely largely on the LLM’s in-context decision-making ability or manually designed rules. As a result, they may struggle with complex memory updates and make incorrect decisions. For example, as shown in Yan et al. (2026)’s work, when a user first says “I adopted a dog named Buddy” and later adds “I adopted another dog named Scout”, a vanilla memory system may incorrectly interpret the new information as a contradiction and perform a **DELETE+ADD** operation, overwriting the original memory. In contrast, a trained memory manager can recognize that the two statements are complementary and perform an **UPDATE** operation to consolidate the information into a more complete memory: “The user adopted two dogs, Buddy and Scout”.

One promising approach to improve memory management is to optimize memory operations with RL (Yan et al., 2026). The key idea is to make memory management itself a learnable decision-making process. Given an extracted memory x^{mem} and an existing memory bank $\mathcal{M}_{\text{old}}$, the memory manager acts as a policy that selects a memory operation and generates the updated memory content:

$$(\text{op}, m') \sim \pi_{\theta}(\cdot \mid x^{\text{mem}}, \mathcal{M}_{\text{old}}) \quad (13)$$

where op denotes the selected memory operation from the operation set $\mathcal{A}^m$ and m' denotes the updated memory content. After applying the selected operation, the updated memory bank is provided to the agent for downstream task solving. By optimizing the memory manager with task-level feedback, the agent can learn when and how to update its memory based on the usefulness of stored experience. Figure 5 illustrates the training process of a memory manager with RL.

During optimization, the reward is defined according to the final task performance. If the updated memory helps the agent produce a correct answer, the memory manager receives a positive reward; otherwise, it receives a lower reward. A simple reward function can be defined as:

$$R_{\text{answer}} = \text{EM}(y_{\text{pred}}, y_{\text{gold}}) \quad (14)$$

where $\text{EM}(\cdot)$ denotes the Exact Match function, y_{pred} denotes the predicted answer, and y_{gold} denotes the ground-truth answer. This design avoids the need to manually annotate individual memory operations.

Instead, the memory manager learns which operations are beneficial by directly optimizing their impact on downstream task performance.

After optimizing the memory manager, the agent can dynamically maintain its memory during future interactions. However, learning effective memory operations is only the first step toward experience-driven agent improvement. A key challenge is how to evaluate the long-term utility of accumulated memories. Existing memory optimization methods typically define rewards based on downstream task performance, i.e., whether the retrieved memories help the agent solve the current task. However, the usefulness of a memory is not always reflected immediately (Xu et al., 2025). For example, a failure case collected from one interaction may not improve the current response, but it can help the agent avoid similar mistakes in future tasks. This indicates that memory can serve as an important component of agent learning, beyond simply storing past information.

Beyond optimizing memory update operations, another important question is how to construct and organize memories from accumulated experiences. Instead of treating memory construction as a fixed preprocessing step, we can enable an agent to learn effective memory construction strategies through RL. In this way, a straightforward approach is to formulate memory construction as a sequential decision-making process, where the agent learns how to select, organize, and transform interaction experiences into useful memories (Wang et al., 2025b).

6.4.2 Skill Optimization

A skill represents a higher-level abstraction of agentic experience. Different from memory, which focuses on storing useful information extracted from previous interactions, a skill aims to summarize recurring patterns across multiple experiences and transform them into reusable capabilities. Specifically, a skill can be viewed as a structured instruction package that describes when the skill should be invoked, what workflow it should follow, which tools are required, and how the final outcome should be verified. For example, a reasoning-oriented skill may be represented as follows:

Name: Math-Logic-Reasoning

Description: Solve complex mathematical, logical, tabular, and constraint reasoning tasks. Use when a task requires multi-step arithmetic, symbolic reasoning, calculator-backed verification, table analysis, optimization, scheduling or allocation constraints, Pareto trade-offs, proof checking, or translating word problems into equations, code, formulas, or executable checks.

Math Logic Reasoning

Use this skill for reasoning tasks where a wrong intermediate assumption can silently break the final answer. Prefer a compact formal model plus executable verification over unsupported mental arithmetic.

Workflow

1. Restate the target quantity, decision, or claim.
2. Extract givens, units, constraints, and hidden assumptions.
3. Convert the problem into equations, inequalities, tables, spreadsheet formulas, graphs, search spaces, constraint satisfaction problems, or short verification scripts.

...

Tool Choices

1. Use Python as a calculator for multi-step arithmetic, combinatorics, simulations, optimization search, or exact rational checks.
2. Use Fraction, Decimal, or sympy when exactness matters.
3. Use pandas for complex tables, joins, grouped summaries, rankings, or data-cleaning logic.

...

Verification Checklist

1. Recalculate the final numeric answer using a second method when feasible.
2. Test boundary cases: zero, negative values, equality limits, duplicate entries, and empty sets.
3. Confirm units after every transformation.

...

Output Style

Give the answer first when possible. Then include a concise derivation, the checked constraints, and any caveats.

In practice, the agent is provided with descriptions of all available skills in the prompt and learns to select appropriate skills based on the current task. These selected skills serve as high-level behavioral templates that guide the agent’s planning and execution process. For example,

Available Skills:Skill 1: Web Search

This skill enables the agent to retrieve and synthesize external information when the task requires up-to-date or domain-specific knowledge ...

Skill 2: Math-Logic-Reasoning

Solve complex mathematical, logical, tabular, and constraint reasoning tasks. Use when a task requires multi-step arithmetic, symbolic reasoning ...

{More skill descriptions}

User Query: Solve the equation $2x + 5 = 13$.

Selected Skill: Math-Logic-Reasoning

A straightforward approach to constructing skills for an agent is to manually write them according to specific task requirements. However, there can be many different ways to design a skill for the same task, similar to the process of writing prompts for LLMs. For example, one may define a skill with detailed instructions,

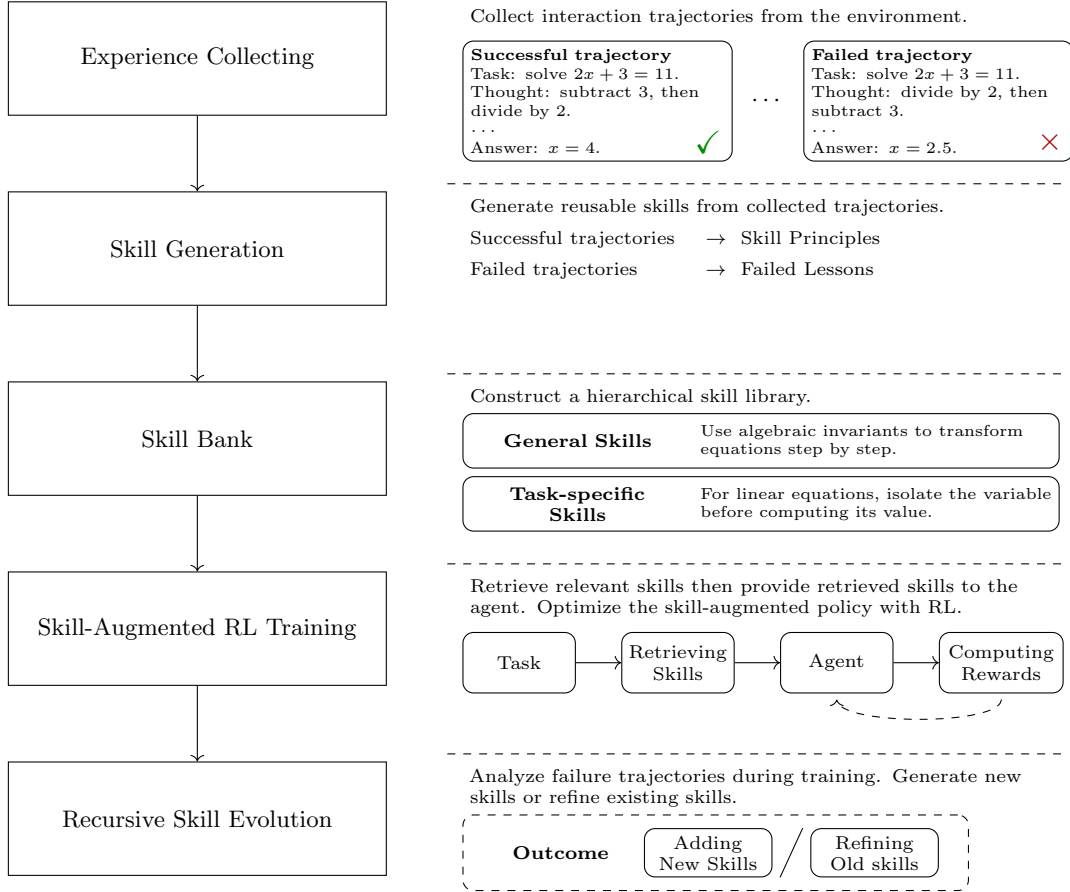


Figure 6: Illustration of SkillRL (Xia et al., 2026). SkillRL maintains a pool of agentic trajectories and initially abstracts reusable skills from these trajectories. The extracted skills are then used to guide RL training. As the agent improves through RL training, it generates higher-quality trajectories, which are further used to refine the skill bank.

including explicit workflows and execution procedures, while another may provide only high-level guidance and allow the agent to determine the detailed execution strategy. Such variations in skill design can significantly affect agent performance (Yu et al., 2026).

One limitation of manual skill construction is that the quality and diversity largely depend on human experience. Therefore, if we want LLMs to handle a broad range of tasks, relying on human-annotated data for LLM fine-tuning is often inefficient. To address this limitation, an alternative approach is to automatically generate skills from agentic experiences. For example, we can prompt an LLM to summarize successful trajectories and extract reusable skills. These generated skills can then be added to the agent’s skill bank and reused in future tasks.

The above way of generating skills often suffers from quality issues. First, a general LLM does not inherently know how to design effective skills without additional optimization. Second, the skill generator may optimize for the wrong objective because skill generation and skill utilization involve different contexts. Specifically, an LLM may consider a generated skill effective because it satisfies the instructions provided in the prompt, while another agent may fail to use this skill effectively in real-world tasks. As a result, skill generation should not only focus on producing well-structured skills, but also consider whether these skills can actually improve agent performance during task execution. Recent work has turned to RL to achieve this goal, where skill generation is optimized based on the actual performance of generated skills.

Here we consider **SkillRL** as an example to illustrate how to optimize skill generation through RL (Xia et al., 2026). The idea is that instead of directly generating skills from stored trajectories, we can first abstract reusable skills from existing trajectories and use them to guide agentic RL training. During the training process, the agent continuously discovers new skills and updates the skill bank, enabling the joint evolution of the agent and its skills. Figure 6 shows a schematic illustration of SkillRL. Here we give a brief outline of the key steps involved.

- Initially, we collect interaction trajectories from environments. These trajectories can include both successful and failed experiences, which provide diverse signals for skill discovery and optimization. Successful trajectories reveal effective behaviors, while failed trajectories help identify potential weaknesses and improvement opportunities.
- The collected trajectories are then used to extract reusable skills, which initialize the skill bank. Considering that agent capabilities usually include both general problem-solving strategies and task-specific procedures, we can organize skills at different levels of abstraction. Specifically, a hierarchical skill library can be constructed, where a general skill bank stores transferable skills learned across different tasks to improve generalization, while a task-specific skill bank preserves specialized strategies for solving particular tasks.
- After initializing the skill bank, the agent retrieves relevant skills to guide the RL training process. Specifically, given task x , the agent selects suitable skills from the skill bank $\mathcal{S}$:

$$\mathcal{S}^* = \text{TopK}_{s_i \in \mathcal{S}} \text{Score}(x, s_i) \quad (15)$$

where $\text{Score}(x, s_i)$ measures the relevance between the current task and each skill, and $\mathcal{S}^*$ denotes the selected skill set. The selected skills are then incorporated into the agent’s decision-making process and used to guide RL optimization. These skills provide high-level behavioral priors, such as task decomposition and execution workflows, which help the agent explore more effective behaviors and improve the efficiency of RL training.

- Through interaction with the environment, the agent collects new trajectories. These trajectories are further analyzed to discover new skills or refine existing ones. In this way, the skill bank evolves together with the agent policy: the skill library provides prior knowledge to accelerate RL training, while improved trajectories continuously enrich and refine the skill library.

By repeating the above process, the agent can improve its skill library through accumulated experiences. Specifically, newly collected trajectories provide additional evidence for discovering new skills or refining existing ones, while the updated skill library further guides future RL training. This creates an iterative learning process in which the agent and its skills co-evolve over time. From this perspective, skill optimization provides a practical way to transform low-level interaction experiences into reusable capabilities, and has become an important approach for enabling agents to continuously learn from their own experiences (Wang et al., 2026b; Fu et al., 2026).

6.4.3 Trajectory Refinement

Different from memory management and skill optimization, which focus on storing and abstracting experiences, trajectory refinement directly improves agent behaviors by revising generated trajectories. This idea is motivated by the self-refinement capability of LLMs. As discussed in Section ??, reasoning-enhanced RL training enables LLMs to evaluate their intermediate reasoning processes and revise incorrect solutions. For example, when solving mathematical problems, an LLM can verify intermediate results after each reasoning step and adjust its solution trajectory based on the verification outcomes.

A natural extension is to apply this self-refinement capability to LLM-based agents. During interaction with environments, agents continuously receive external feedback, such as tool execution results or environment states. Instead of blindly following the initial trajectory, agents can leverage these feedback signals to identify mistakes, revise previous decisions, and generate improved trajectories. In this way, agents can

learn from their own interaction experiences by continuously refining their behaviors. One commonly used trajectory refinement approach is Reflexion (Shinn et al., 2023). The basic idea is to introduce an explicit reflection process into agent execution. Given a task, the agent first generates an initial trajectory and receives feedback from an evaluator. The agent then analyzes the feedback, summarizes potential errors or improvement strategies, and generates a reflection. Based on this reflection, the agent produces an improved trajectory in subsequent attempts. Here, we take mathematical reasoning as an example.

Task: Solve the equation $3x + 7 = 22$.

Initial Trajectory:

Thought: I need to isolate x by moving the constant term to the other side.

Action: Subtract 7 from both sides:

$$3x + 7 - 7 = 22 - 7$$

Observation: The equation becomes:

$$3x = 15$$

Thought: I need to divide both sides by 3 to obtain x .

Action: Calculate:

$$x = \frac{15}{3} = 4$$

Answer: $x = 4$.

Feedback: The answer is incorrect. This is because: $\frac{15}{3} = 5 \neq 4$.

Reflection: The previous trajectory failed because I made an error during intermediate calculation. For future tasks, I should carefully verify arithmetic operations and check the final answer by substituting it back into the original equation.

Refined Trajectory:

Thought: I need to isolate x and verify the final solution.

Action: Subtract 7 from both sides:

$$3x = 22 - 7 = 15$$

Action: Divide both sides by 3:

$$x = \frac{15}{3} = 5$$

Observation: Verify the solution:

$$3 \times 5 + 7 = 22$$

Answer: $x = 5$.

It is worth noting that we can obtain feedback from either internal evaluation or external evaluators. For example, we can use rule-based verifiers, LLM-based evaluators, or binary reward signals to provide feedback to the agent. Regardless of the feedback source or format, we need to convert it into a language-based representation, enabling the agent to understand errors and refine future behaviors. Here, inspired by the step-level feedback discussed in Section ??, we can further consider whether providing fine-grained feedback at each decision step can help agents generate better-refined trajectories. Compared with outcome-level feedback, step-level feedback provides more precise guidance about where and why the agent makes mistakes (Liu et al., 2024; Wang et al., 2025a). For example, when an agent fails to solve a mathematical reasoning problem, a binary reward only indicates that the final answer is incorrect, but it does not reveal which reasoning step contains the error or how the agent should revise its solution. However, obtaining step-level feedback is often expensive and challenging, as it requires evaluating intermediate decisions throughout the trajectory. To address this challenge, recent studies explore using methods such as MCTS to automatically construct step-level feedback signals (Yuan et al., 2025).

The above approaches mainly rely on prompting to enable trajectory refinement. Such approaches still face several limitations. First, their performance is bounded by the intrinsic refinement ability of the LLM. A pre-trained LLM may not naturally know how to interpret feedback and effectively revise its trajectory without additional optimization. Second, prompt-based refinement mainly improves the current trajectory at inference time, which limits its ability to accumulate and transfer experience across tasks. As discussed in Sections 6.4.1 and 6.4.2, failed trajectories contain valuable experiences about agent weaknesses and can provide useful learning signals for improving future behaviors. However, prompting-based approaches only leverage these experiences within the current interaction context. To address these limitations, recent studies explore training agents to acquire trajectory refinement abilities (Fu et al., 2025), either by updating model parameters or optimizing refinement behaviors based on collected experiences (Song et al., 2024). Interested readers can refer to these works for more information.

Although trajectory refinement does not explicitly update model parameters like traditional RL, it shares the fundamental principle of RL: improving agent behaviors based on feedback collected from interactions. From the perspective of in-context learning, the feedback obtained during trajectory refinement can be viewed as a temporary reward signal that modifies the agent’s subsequent decision-making process. Instead of updating policy parameters, the agent updates its context with reflections, which potentially modifies the policy used for future actions within the current interaction. Therefore, trajectory refinement can be regarded as an in-context form of policy improvement. This perspective is closely related to the emerging research direction of *in-context RL* (namely ICRL), which studies how agents can adapt their behaviors through interaction feedback without explicit parameter updates (Monea et al., 2024).

References

- Shihao Cai, Runnan Fang, Jialong Wu, Baixuan Li, Xinyu Wang, Yong Jiang, Liangcai Su, Liwen Zhang, Wenbiao Yin, Zhen Zhang, Fuli Feng, Pengjun Xie, and Xiaobin Wang. Autoforge: Automated environment synthesis for agentic reinforcement learning, 2025. URL <https://arxiv.org/abs/2512.22857>.
- Kaiyan Chang, Songcheng Xu, Chenglong Wang, Yingfeng Luo, Xiaoqian Liu, Tong Xiao, and Jingbo Zhu. Efficient prompting methods for large language models: A survey, 2024. URL <https://arxiv.org/abs/2404.01077>.
- Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. Fireact: Toward language agent fine-tuning, 2023. URL <https://arxiv.org/abs/2310.05915>.
- Jianming Chen, Yawen Wang, Junjie Wang, Xiaofei Xie, Shoubin Li, Qing Wang, and Fanjiang Xu. Where did it go wrong? capability-oriented failure attribution for vision-and-language navigation agents. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 28132–28150, San Diego, California, United States, 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. URL <https://aclanthology.org/2026.findings-acl.1402/>.
- Mingyang Chen, Linzhuang Sun, Tianpeng Li, Haoze Sun, Yijie Zhou, Chenzheng Zhu, Haofen Wang, Jeff Z. Pan, Wen Zhang, Huajun Chen, Fan Yang, Zenan Zhou, and Weipeng Chen. Research: Learning to reason with search for llms via reinforcement learning. In Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025. URL http://papers.nips.cc/paper_files/paper/2025/hash/7b3a0d3a0864b66f229c0a9c84f9e950-Abstract-Conference.html.
- Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. Mem0: Building production-ready ai agents with scalable long-term memory, 2025. URL <https://arxiv.org/abs/2504.19413>.
- DeepSeek-AI. Deepseek-v3.2: Pushing the frontier of open large language models, 2025. URL <https://arxiv.org/abs/2512.02556>.
- Runnan Fang, Shihao Cai, Baixuan Li, Jialong Wu, Guangyu Li, Wenbiao Yin, Xinyu Wang, Xiaobin Wang, Liangcai Su, Zhen Zhang, Shibin Wu, Zhengwei Tao, Yong Jiang, Pengjun Xie, Ningyu Zhang, Fei Huang, Wentao Zhang, and Jingren Zhou. Towards general agentic intelligence via environment scaling. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 17610–17621, San Diego, California, United States, 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. URL <https://aclanthology.org/2026.findings-acl.872/>.
- Jiazhan Feng, Shijue Huang, Xingwei Qu, Ge Zhang, Yujia Qin, Baoquan Zhong, Chengquan Jiang, Jinxin Chi, and Wanjuan Zhong. Retool: Reinforcement learning for strategic tool use in llms, 2025. URL <https://arxiv.org/abs/2504.11536>.
- Dayuan Fu, Keqing He, Yejie Wang, Wentao Hong, Zhuoma Gongque, Weihao Zeng, Wei Wang, Jingang Wang, Xunliang Cai, and Weiran Xu. Agentrefine: Enhancing agent generalization through refinement tuning. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL <https://openreview.net/forum?id=FDimWzmcWn>.
- Zenghuang Fu, Zhaoyang Li, Qiuyuan Ai, Haoyu Wu, Minghui Wu, Chenxu Zhao, Ante Wang, Guannan He, and Changwei Wang. Self-play meets skill evolution: Self-evolving search agents that pose, solve, and remember, 2026. URL <https://arxiv.org/abs/2607.29468>.

- Mengkang Hu, Pu Zhao, Can Xu, Qingfeng Sun, Jian-Guang Lou, Qingwei Lin, Ping Luo, and Saravan Rajmohan. Agentgen: Enhancing planning abilities for large language model based agent via environment and task generation. In Yizhou Sun, Flavio Chierichetti, Hady W. Lauw, Claudia Perlich, Wee Hyong Tok, and Andrew Tomkins (eds.), *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining, V.1, KDD 2025, Toronto, ON, Canada, August 3-7, 2025*, pp. 496–507. ACM, 2025. URL <https://doi.org/10.1145/3690624.3709321>.
- Yifu Huo, Shunjie Xing, Chenglong Wang, Peinan Feng, Qiaozhi He, Yan Ding, Anxiang Ma, Yuxin Gao, Tongran Liu, Tong Xiao, and Jingbo Zhu. Learning from environmental feedback: Credit assignment across multiple timescales for agentic reinforcement learning, 2026. URL <https://arxiv.org/abs/2608.08255>.
- Bowen Jin, Hansi Zeng, Zhenrui Yue, Dong Wang, Hamed Zamani, and Jiawei Han. Search-r1: Training llms to reason and leverage search engines with reinforcement learning, 2025. URL <https://arxiv.org/abs/2503.09516>.
- Mojtaba Komeili, Kurt Shuster, and Jason Weston. Internet-augmented dialogue generation. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (eds.), *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8460–8478, Dublin, Ireland, 2022. Association for Computational Linguistics. URL <https://aclanthology.org/2022.acl-long.579/>.
- Joonho Lee, Marko Bjelonic, Alexander Reske, Lorenz Wellhausen, Takahiro Miki, and Marco Hutter. Learning robust autonomous navigation and locomotion for wheeled-legged robots. *Science Robotics*, 9 (89):eadi9641, 2024.
- Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. API-bank: A comprehensive benchmark for tool-augmented LLMs. In Houda Bouamor, Juan Pino, and Kalika Bali (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 3102–3116, Singapore, 2023. Association for Computational Linguistics. URL <https://aclanthology.org/2023.emnlp-main.187/>.
- Yixia Li, Hongru Wang, Jiahao Qiu, Zhenfei Yin, Dongdong Zhang, Cheng Qian, Zeping Li, Xiaoteng Ma, Guanhua Chen, and Heng Ji. From word to world: Can large language models be implicit text-based world models? In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 8084–8111, San Diego, California, United States, 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. URL <https://aclanthology.org/2026.acl-long.366/>.
- Zhiwei Li, Yong Hu, and Wenqing Wang. Encouraging good processes without the need for good answers: Reinforcement learning for LLM agent planning. In Saloni Potdar, Lina Rojas-Barahona, and Sebastien Montella (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pp. 1654–1666, Suzhou (China), 2025. Association for Computational Linguistics. ISBN 979-8-89176-333-3. URL <https://aclanthology.org/2025.emnlp-industry.116/>.
- Xiaoqian Liu, Ke Wang, Yuchuan Wu, Fei Huang, Yongbin Li, Jianbin Jiao, and Junge Zhang. Agentic reinforcement learning with implicit step rewards. In *International Conference on Learning Representations*, volume 2026, pp. 129271–129291, 2026.
- Zhiwei Liu, Weiran Yao, Jianguo Zhang, Rithesh Murthy, Liangwei Yang, Zuxin Liu, Tian Lan, Ming Zhu, Juntao Tan, Shirley Kokane, Thai Hoang, Juan Carlos Niebles, Shelby Heinecke, Huan Wang, Silvio Savarese, and Caiming Xiong. PRACT: Optimizing principled reasoning and acting of LLM agent. In Libby Barak and Malihe Alikhani (eds.), *Proceedings of the 28th Conference on Computational Natural Language Learning*, pp. 442–446, Miami, FL, USA, 2024. Association for Computational Linguistics. URL <https://aclanthology.org/2024.conll-1.33/>.
- Shayne Longpre, Le Hou, Tu Vu, Albert Webson, Hyung Won Chung, Yi Tay, Denny Zhou, Quoc V. Le, Barret Zoph, Jason Wei, and Adam Roberts. The flan collection: Designing data and methods for

- effective instruction tuning. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (eds.), *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA*, volume 202 of *Proceedings of Machine Learning Research*, pp. 22631–22648. PMLR, 2023. URL <https://proceedings.mlr.press/v202/longpre23a.html>.
- Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013. URL <https://arxiv.org/abs/1312.5602>.
- Giovanni Monea, Antoine Bosselut, Kianté Brantley, and Yoav Artzi. Llms are in-context reinforcement learners. 2024.
- Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. Gorilla: Large language model connected with massive apis. In Amir Globersons, Lester Mackey, Danielle Belgrave, Angela Fan, Ulrich Paquet, Jakub M. Tomczak, and Cheng Zhang (eds.), *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, NeurIPS 2024, Vancouver, BC, Canada, December 10 - 15, 2024*, 2024. URL http://papers.nips.cc/paper_files/paper/2024/hash/e4c61f578ff07830f5c37378dd3ecb0d-Abstract-Conference.html.
- Cheng Qian, Emre Can Acikgoz, Qi He, Hongru Wang, Xiusi Chen, Dilek Hakkani-Tur, Gokhan Tur, and Heng Ji. Toolrl: Reward is all tool learning needs. In Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025. URL http://papers.nips.cc/paper_files/paper/2025/hash/97c5b2707228e7e3fb67e4ecc2e0e607-Abstract-Conference.html.
- 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. Toolllm: Facilitating large language models to master 16000+ real-world apis. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL <https://openreview.net/forum?id=dHng200Jjr>.
- Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/d842425e4bf79ba039352da0f658a906-Abstract-Conference.html.
- Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: language agents with verbal reinforcement learning. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/1b44b878bb782e6954cd888628510e90-Abstract-Conference.html.
- David Silver and Richard S Sutton. Welcome to the era of experience. *Google AI*, 1:11, 2025.
- David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676):354–359, 2017.

- Joykirat Singh, Raghav Magazine, Yash Pandya, and Akshay Nambi. Agentic reasoning and tool integration for llms via reinforcement learning, 2025a. URL <https://arxiv.org/abs/2505.01441>.
- Joykirat Singh, Yash Pandya, Pranav Vajreshwari, Raghav Magazine, and Akshay Nambi. Agentic reasoning and tool integration for llms via reinforcement learning. In *First Workshop on Foundations of Reasoning in Language Models*, 2025b.
- Joykirat Singh, Yash Pandya, Pranav Vajreshwari, Raghav Magazine, and Akshay Nambi. Agentic reasoning and tool integration for llms via reinforcement learning. In *First Workshop on Foundations of Reasoning in Language Models*, 2025c.
- Xiaoshuai Song, Haofei Chang, Guanting Dong, Yutao Zhu, Ji-Rong Wen, and Zhicheng Dou. EnvScaler: Scaling tool-interactive environments for LLM agent via programmatic synthesis. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Findings of the Association for Computational Linguistics: ACL 2026*, pp. 8326–8357, San Diego, California, United States, 2026. Association for Computational Linguistics. ISBN 979-8-89176-395-1. URL <https://aclanthology.org/2026.findings-acl.407/>.
- Yifan Song, Da Yin, Xiang Yue, Jie Huang, Sujian Li, and Bill Yuchen Lin. Trial and error: Exploration-based trajectory optimization of LLM agents. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 7584–7600, Bangkok, Thailand, 2024. Association for Computational Linguistics. URL <https://aclanthology.org/2024.acl-long.409/>.
- Michael Sullivan, Mareike Hartmann, and Alexander Koller. Procedural environment generation for tool-use agents. In Christos Christodoulopoulos, Tanmoy Chakraborty, Carolyn Rose, and Violet Peng (eds.), *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 18544–18562, Suzhou, China, 2025. Association for Computational Linguistics. ISBN 979-8-89176-332-6. URL <https://aclanthology.org/2025.emnlp-main.936/>.
- Xueqiao Sun, Xiaohan Wang, Ludwig Schmidt, Serena Yeung-Levy, and Yuhui Zhang. Learning from failure: Inference-time self-improvement for computer-use agents, 2026. URL <https://arxiv.org/abs/2606.31270>.
- Richard Stuart Sutton. *Temporal credit assignment in reinforcement learning*. University of Massachusetts Amherst, 1984.
- Daoyu Wang, Qingchuan Li, Mingyue Cheng, Jie Ouyang, Shuo Yu, Qi Liu, and Enhong Chen. Steppo: Step-aligned policy optimization for agentic reinforcement learning, 2026a. URL <https://arxiv.org/abs/2604.18401>.
- Hanlin Wang, Jian Wang, Chak Tou Leong, and Wenjie Li. STeCa: Step-level trajectory calibration for LLM agent learning. In Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (eds.), *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 11597–11614, Vienna, Austria, 2025a. Association for Computational Linguistics. ISBN 979-8-89176-256-5. URL <https://aclanthology.org/2025.findings-acl.604/>.
- Jiong Xiao Wang, Qiaojing Yan, Yawei Wang, Yijun Tian, Soumya Smruti Mishra, Zhichao Xu, Megha Gandhi, Panpan Xu, and Lin Lee Cheong. Reinforcement learning for self-improving agent with skill library. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1529–1550, San Diego, California, United States, 2026b. Association for Computational Linguistics. ISBN 979-8-89176-390-6. URL <https://aclanthology.org/2026.acl-long.69/>.
- Yu Wang, Ryuichi Takanobu, Zhiqi Liang, Yuzhen Mao, Yuanzhe Hu, Julian McAuley, and Xiaojian Wu. Mem- $\{\alpha\}$: Learning memory construction via reinforcement learning, 2025b. URL <https://arxiv.org/abs/2509.25911>.

- Zhaoyang Wang, Canwen Xu, Boyi Liu, Yite Wang, Siwei Han, Zhewei Yao, Huaxiu Yao, and Yuxiong He. Agent world model: Infinity synthetic environments for agentic reinforcement learning, 2026c. URL <https://arxiv.org/abs/2602.10090>.
- Zhiheng Xi, Chenyang Liao, Guanyu Li, Zhihao Zhang, Wenxiang Chen, Binghai Wang, Senjie Jin, Yuhao Zhou, Jian Guan, Wei Wu, Tao Ji, Tao Gui, Qi Zhang, and Xuanjing Huang. Agentprm: Process reward models for LLM agents via step-wise promise and progress. In Hakim Hacid, Yoelle Maarek, Francesco Bonchi, Ido Guy, and Emine Yilmaz (eds.), *Proceedings of the ACM Web Conference 2026, WWW 2026, Dubai, United Arab Emirates, originally scheduled for April 13-17, 2026, rescheduled for June 29 - July 3, 2026*, pp. 4184–4195. ACM, 2026. URL <https://doi.org/10.1145/3774904.3792551>.
- Peng Xia, Jianwen Chen, Hanyang Wang, Jiaqi Liu, Kaide Zeng, Yu Wang, Siwei Han, Yiyang Zhou, Xujiang Zhao, Haifeng Chen, et al. Skillrl: Evolving agents via recursive skill-augmented reinforcement learning, 2026. URL <https://arxiv.org/abs/2602.08234>.
- Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-mem: Agentic memory for LLM agents. In Danielle Belgrave, Cheng Zhang, Laura N. Montoya, Hsuan-Tien Lin, Razvan Pascanu, Piotr Koniusz, Marzyeh Ghassemi, Nancy Chen, Iván Vladimir Meza Ruíz, and Arturo Loaiza-Bonilla (eds.), *Advances in Neural Information Processing Systems 38: Annual Conference on Neural Information Processing Systems 2025, NeurIPS 2025, San Diego, CA, USA, December 2-7, 2025 / Mexico City, Mexico, November 30 - December 5, 2025*, 2025. URL http://papers.nips.cc/paper_files/paper/2025/hash/19909c36f51abc4856b4560aff3d36d6-Abstract-Conference.html.
- Sikuan Yan, Xiufeng Yang, Zuchao Huang, Ercong Nie, Zifeng Ding, Zonggen Li, Xiaowen Ma, Jinhe Bi, Kristian Kersting, Jeff Z. Pan, Hinrich Schuetze, Volker Tresp, and Yunpu Ma. Memory-r1: Enhancing large language model agents to manage and utilize memories via reinforcement learning. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens (eds.), *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12805–12825, San Diego, California, United States, 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. URL <https://aclanthology.org/2026.acl-long.583/>.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/pdf?id=WE_vluYUL-X.
- Da Yin, Faeze Brahman, Abhilasha Ravichander, Khyathi Chandu, Kai-Wei Chang, Yejin Choi, and Bill Yuchen Lin. Agent lumos: Unified and modular training for open-source language agents. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12380–12403, Bangkok, Thailand, 2024. Association for Computational Linguistics. URL <https://aclanthology.org/2024.acl-long.670/>.
- Jianxiang Yu, Jiapeng Zhu, Bochen Lin, Qier Cui, Zichen Ding, and Xiang Li. Skill is not one-size-fits-all: Model-aware skill alignment for llm agents, 2026. URL <https://arxiv.org/abs/2605.30723>.
- Siyu Yuan, Zehui Chen, Zhiheng Xi, Junjie Ye, Zhengyin Du, and Jiecao Chen. Agent-r: Training language model agents to reflect via iterative self-training, 2025. URL <https://arxiv.org/abs/2501.11425>.
- Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. AgentTuning: Enabling generalized agent abilities for LLMs. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 3053–3077, Bangkok, Thailand, 2024. Association for Computational Linguistics. URL <https://aclanthology.org/2024.findings-acl.181/>.
- Jianguo Zhang, Tian Lan, Rithesh Murthy, Zhiwei Liu, Weiran Yao, Ming Zhu, Juntao Tan, Thai Hoang, Zuxin Liu, Liangwei Yang, et al. Agentohana: Design unified data and training pipeline for effective agent learning, 2024. URL <https://arxiv.org/abs/2402.15506>.

Chunting Zhou, Pengfei Liu, Puxin Xu, Srinivasan Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, Susan Zhang, Gargi Ghosh, Mike Lewis, Luke Zettlemoyer, and Omer Levy. LIMA: less is more for alignment. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/ac662d74829e4407ce1d126477f4a03a-Abstract-Conference.html.

Meng Zhou, Ziyu Liu, Pengwei Sui, Yixuan Li, and Yuk Ying Chung. Learning implicit credit assignment for cooperative multi-agent reinforcement learning. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL <https://proceedings.neurips.cc/paper/2020/hash/8977ecbb8cb82d77fb091c7a7f186163-Abstract.html>.