

Next-Generation Agentic Reinforcement Learning Systems Enable Self-Evolving Agents

Ran Yan^{1,2}, Wei Fu^{1,3}, Jiale Li¹, Shusheng Xu¹, Zhiyu Mei¹, Jiaxuan Gao^{1,3}, Jiarui Zhang^{1,2,3}, Hao Dai¹, Xujie Shen¹, Chuyi He¹, Zhen Pu¹, Jun Mei¹, Zhiyao Lin¹, Haitao Wang¹, Zhiqiang Ding¹, Jiawei Zhang¹, Huaijie Wang^{1,3}, Ruida Xu¹, Youhe Jiang², Yi Wu³, Tongkai Yang¹, Binhang Yuan^{1,2}

¹Ant Group, ²HKUST, ³Tsinghua University

Abstract

Large language model (LLM) agents are rapidly being deployed in production, including coding assistants, customer-support chatbots, and scientific research assistants, yet they remain fundamentally static in large-scale enterprise-level deployment. The LLM weights, system prompts, tool repertoires, and in-context harnesses are frozen at deployment time, and any improvement requires a manual loop of human-curated data collection, offline fine-tuning, modification of the agentic paradigm, and re-deployment. Recent work on self-evolving agents, such as OpenClaw for individual users, indicates that the next leap in agent capability will come from agents that continually learn from their own experience. In this paper, we argue that this vision for self-evolving agent deployment is being held back for enterprise-level large-scale agentic service not by reinforcement learning (RL) algorithms but by agentic online RL systems. Specifically, we argue that current agentic RL systems and the surrounding observability software stack are inadequate along three essential aspects: (i) there is no standardized *agent trajectory data protocol* capable of carrying RL learning signals at step granularity across heterogeneous agent paradigms; (ii) there is no enterprise-grade comprehensive *data proxy* that converts real workloads into governed learning substrates; and (iii) there is no *unified agent evolution control plane* with automatic triggering that decides, on the basis of trajectory statistics, when to update policy model weights or evolve the in-context harness by the corresponding RL algorithm. We argue that the next generation of agentic RL systems must be co-designed around these three pillars, and we sketch concrete architectural commitments, case studies, and counter-arguments to substantiate our position. We further instantiate one scoped branch of our vision through AREAL2.0, showing how existing RL infrastructure can be reorganized into an agent service-oriented online RL loop for policy LLM weight updates from deployed online agent workloads.

 Code: <https://github.com/areal-project/AReal>

1 Introduction

Large language model (LLM) agents are moving from laboratory demonstrations into deployed systems for research assistance, software engineering, and task automation. In fact, such LLM agents should essentially change the unit of deployment for LLM services at enterprise level—the deployed object should no longer be only a LLM that maps a sequence of input tokens to a sequence of output tokens; instead it should increasingly be a long-horizon agent policy embedded in a heterogeneous complicated business environment, where the agent reads files, calls tools, retrieves documents, invokes APIs, updates memory, requests human approval, and accomplishes complicated analytic jobs. This shift creates a *mismatch between how agents are deployed and how agents are improved*: A deployed enterprise agent may perform thousands or millions of tool-mediated interactions, yet its future behavior usually changes only through a manual engineering cycle: operators inspect traces, define new evaluation benchmarks, edit prompts, add tool descriptions, update a LLM through

reinforcement learning (RL), implement a new agentic loop, and redeploy. On the other hand, we have recently witnessed that self-evolving agents are no longer hypothetical—personal agent systems, such as OpenClaw [1], and learning frameworks such as OpenClaw-RL [2], suggest that the next leap in agent capability will come from agents that continually learn from situated experience [3, 4].

We interpret a *self-evolving agent* as a deployed agent whose future behavior can improve as a consequence of its prior situated experience. To be cautious, we do not mean unrestricted recursive self-modification; instead, we mean a bounded closed loop in which each user’s interaction trajectories with the agent can be observed, redacted, verified, attributed, and converted into one of several governed updates: a memory insertion, a skill patch, a harness edit, a tool-schema modification, or an on-policy RL update. Recent agents built from trajectory reflection [5], memory extraction [6], context/harness edit [7], skill libraries [4], and self-feedback [2] already demonstrate that execution traces can be turned into reusable guidance.

However, current self-evolving agent examples remain closer to individual-assistant or algorithm-specific self-improvement than to general enterprise-scale self-evolving agent services. For example, OpenClaw [1] is compelling because it reframes the assistant as a persistent, user-owned agent that can perform tasks across inboxes, calendars, flights, and messaging apps, but this is not the same as a governed enterprise learning substrate spanning many teams, tenants, workflows, and compliance boundaries. On the other hand, an enterprise coding agent should learn from issue-resolution traces across teams while respecting repository permissions, license boundaries, proprietary code restrictions, and audit requirements; a customer support agent should learn from escalations, refunds, satisfaction signals, and policy updates while avoiding the leakage of customer data; a scientific agent must learn from failed experiments and literature-search trajectories while preserving provenance, reproducibility, and laboratory constraints. We believe that at the enterprise level, the missing component is not a larger model, a cleverer prompt, or even a specific RL algorithm; it is an enterprise-level system substrate that supports the smooth transfer from agentic interaction trajectories to self-evolving online learning. The current research focuses on agents that often prioritize improvements to better models, better prompting, and better toolkit integration, but this framing misses the system challenges for real-world large-scale deployment.

Self-evolving, agentic RL should therefore be explored not only as an algorithmic family but also as a system discipline for design and implementation: how to capture experience, transform it into credit-assignable data, select an evolution mechanism, and deploy the resulting update appropriately. In this paper, we argue that *self-evolving enterprise agents require next-generation online agentic RL systems*, and that such systems must have three co-designed pillars, including (i) a standardized, vendor-neutral, RL-grade agent trajectory data protocol capable of carrying step-level learning signals across heterogeneous agent paradigms; (ii) an enterprise-grade data proxy that converts real agentic workloads into governed, replayable learning corpus; (iii) an unified agent evolution control plane that automatically decides, from trajectory statistics and operational constraints, whether to update memory, patch a skill, edit a harness, or update model weights through post-training RL. Concretely, we enumerate these three pillars below:

First, self-evolving agents require a standardized *agent trajectory data protocol* (ATDP) that makes the deployed agent experience learnable rather than merely observable. Existing agent logs typically record prompts, completions, tool calls, latency, errors, and token usage; such traces are useful for debugging but insufficient for online agentic RL. A learning-ready trajectory must instead preserve the decision process at step granularity: the observation available to the agent, the relevant internal or harness state, the chosen action, the action outcome, late-arriving reward or critique signals, and metadata such as model version, tool schema, tenant, cost, and governance status. The purpose of ATDP is therefore to turn heterogeneous agent executions into typed, auditable, replayable, and credit-assignable event sequences. This protocol should be vendor-neutral and framework-agnostic, support delayed feedback and reward augmentation, preserve versioned provenance for counterfactual replay, and encode privacy, access-control, retention, and training-eligibility fields from the beginning. Without such a protocol, enterprise agents may generate vast amounts of interaction data, but the data will remain too incomplete, non-reproducible, or unsafe to serve as the substrate for self-evolution.

Second, self-evolving agents require an enterprise-grade *agentic data proxy* that captures ATDP trajectories from real production workloads across models, tools, memory systems, retrieval systems, and human-feedback channels. ATDP specifies what must be represented; the data proxy specifies how those representations are

produced in heterogeneous enterprise environments. Because deployed agents will not be built on a single framework, LLM provider, tool stack, or orchestration paradigm, the data proxy must intercept at stable execution boundaries: LLM calls, tool invocations, retrieval calls, memory reads and writes, file or browser actions, approval events, user corrections, and final task outcomes. Its role is not simply to export traces but to convert production work into governed learning material by redacting sensitive fields, enforcing access control and retention policies, attaching provenance, harvesting weak and delayed reward signals, and persisting trajectories in a form suitable for replay and training. Crucially, the data proxy must distinguish deterministic replay, approximate replay, and non-replayable events, because an enterprise learning system must be able to ask not only what happened, but whether the agent would have succeeded under a different prompt, model checkpoint, memory item, retrieval policy, or tool schema. In this sense, the data proxy serves as the first operational safety boundary of the self-evolving RL loop.

Third, self-evolving agents require a *unified agent evolution control plane* that decides when, where, and how agent behavior should change. Self-evolution should not be equated with blindly updating model weights. A deployed agent is a composite policy consisting of a base LLM, an in-context harness, memory, tools, and guardrails; different failures require different intervention surfaces. Recurring missing facts may call for memory insertion, tool-routing failures may call for harness or schema edits, reusable procedural failures may call for skill patches, and broad failures that persist across tenants, tasks, and tool configurations may justify model-weight updates through supervised fine-tuning, preference optimization, on-policy RL, process-reward learning, or distillation. The control plane should therefore treat self-evolution as a governed decision problem: given a window of ATDP trajectories, trajectory statistics, evaluator scores, user-correction rates, tool-failure clusters, cost signals, safety constraints, and distribution-drift indicators, it selects among memory updates, skill updates, harness edits, tool-schema changes, policy updates, rollback, or no-op. Each selected intervention must pass through replay-first evaluation, offline regression tests, tenant-aware safety checks, and versioned rollback. The control plane is thus the mechanism that turns captured experience into staged, auditable, and automatically triggered agent improvement, making enterprise-scale self-evolving agents possible.

Lastly, to ground this vision in a concrete system prototype, we instantiate one deliberately scoped branch of the proposed architecture in AREAL2.0. Rather than claiming to realize the full self-evolving agent substrate, AREAL2.0 focuses on online policy LLM weight updates as a representative evolution path. This prototype shows how an existing RL framework (i.e., AREAL [8]) can be reorganized from an offline post-training system into an agent service-oriented online RL loop: deployed agent services can redirect their LLM inference calls to AREAL2.0-managed agent-compute workers, while the resulting interaction trajectories are captured and consumed by the RL training pipeline. This design choice lets us explore the practical integration between agent serving, trajectory collection, and policy optimization, while leaving the broader multi-surface evolution problem—including memory updates, skill patches, harness edits, tool-schema evolution, replay-based governance, and automatic intervention selection—as the complete substrate system developing agenda.

2 Related Work

RL for LLM post-training. Reinforcement learning has been the state-of-the-art for LLM post-training [9]. RLHF demonstrated that fine-tuning with human preferences can improve instruction following [10]. Constitutional AI and RLAIF showed that AI-generated critiques and preference signals can substitute for some forms of human labeling [11]. PPO remains a foundational policy-gradient method for constrained policy updates [12], while DPO reframes preference optimization as a simpler classification-style objective that avoids explicit reward-model training and online RL sampling [13]. More recently, reasoning-oriented RL has renewed interest in online and outcome-driven optimization. DeepSeek-R1 [14] reported that reasoning behavior can be incentivized through RL, including emergent self-reflection and strategy adaptation.

Self-evolving agents. Self-evolving agents have emerged as a new paradigm for agentic workflows [15]. ReAct [16] showed that LLMs can interleave reasoning traces and actions, giving rise to trajectories that are interpretable and tool-mediated rather than single-turn completions. Reflexion [5] converted task feedback into linguistic reflections stored in memory. Memento-Skills [6] makes skills first-class evolving memory artifacts and updates them through a read-write reflective learning process. OpenClaw-RL [2] focuses on personalized online signals from user interactions. These paradigms demonstrate the plausibility of learning

from experience without always updating policy LLM weights as a new online agentic RL paradigm. The field, therefore, already has many mechanisms for “what might evolve”: memory [6], skills [4], prompts [17], search tool interfaces [18], and model weights. What the current landscape misses is an online RL system substrate for deciding, from deployed trajectories, which mechanism should evolve and how the resulting update should be evaluated, governed, and deployed.

RL systems. Recent work has optimized systems to improve the efficiency of RL-based post-training for LLMs [19–21]. HybridFlow/verl [22] improves RL training throughput through a hybrid-controller execution model, a 3D-HybridEngine for zero-redundancy actor-model resharding, and automatic GPU placement. StreamRL [23] targets disaggregated RL training by mitigating pipeline bubbles caused by stage dependencies and skewness bubbles caused by long-tail generation lengths, while enabling heterogeneous and elastic resource allocation. AsyncFlow [24] introduces asynchronous streaming dataflow and delayed parameter synchronization between rollout generation and policy updates, reducing hardware idling under bounded staleness. AReal [8] further decouples rollout generation from policy optimization through fully asynchronous execution, using staleness-aware training and a decoupled RL objective to maintain training stability. However, these systems are not sufficient to support self-evolving agents via online reinforcement learning.

Agent and RL data protocol. Standardization has recently become an explicit research axis for both agent systems and RL datasets. Agent interoperability protocols such as MCP [25] and A2A [26] standardize how agents discover tools, access external context, and communicate across heterogeneous providers, while recent surveys classify these protocols primarily around context access, inter-agent coordination, security, scalability, and latency [27]. In parallel, RL data infrastructures such as D4RL [28] and RLDS [29] have shown the importance of standardized trajectory datasets for offline RL, imitation learning, replay, annotation, and reproducible benchmarking. Most directly related, Agent Data Protocol (ADP) proposes a lightweight “interlingua” for unifying heterogeneous LLM-agent datasets across tool use, browsing, coding, and general agentic workflows, demonstrating that standardized agent trajectories can improve scalable supervised fine-tuning [30]. However, these efforts do not target the problem addressed in this paper: deployed self-evolving agents require not only interoperability or offline dataset conversion, but an RL trajectory protocol that preserves step-level causal context, delayed reward signals, action outcomes, harness and tool versions, governance metadata, replay boundaries, and learning eligibility under enterprise constraints.

3 Agent Trajectory Data Protocol

In order to enable self-evolving agents, the first question to ask is:

What should be the right data abstraction that will be captured for learning?

To answer this question, our first step is to build a standardized agent trajectory data protocol (ATDP) capable of carrying RL-step-grade signals at step granularity across heterogeneous agentic workflows. We make the following key statement about this pillar of self-evolving agents:

Key claim (i): *Self-evolving agents require a standardized, vendor-neutral, RL-step-grade trajectory data protocol that captures the full lifecycle of agent decision process, including elements that current observability schemas usually discard.*

A prototype formulation. A trajectory noted by τ under ATDP is a typed event sequence defined as:

$$\tau = (\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_T).$$

And any intermediate step $\mathbf{e}_t$ is defined as:

$$\mathbf{e}_t = \langle \mathbf{o}_t, \mathbf{h}_t, \mathbf{a}_t, \mathbf{y}_t, r_t, \mathbf{m}_t \rangle$$

Where $\mathbf{o}_t$ is the *observable status* (e.g., tool outputs, retrieval snippets, user message, environment state); $\mathbf{h}_t$ is the *hidden internal status* (e.g., followed plan, scratchpad, confidence, reasoning summary); $\mathbf{a}_t$ is the chosen *action* (e.g., tool call with typed argument schema, message in terms of generated tokens, code edit, memory

update); y_t is the *action outcome* for a_t (e.g., tool return, user accept/edit/retry/delete, or exit code); r_t is the reward signal (e.g., binary outcome, scalar score, natural language critique, or implicit signal extracted from y_t); m_t is the relevant *metadata* (e.g., latency, tokens, cost, tenant, session, harness fingerprint, model id). Note that the schema reduces to a standard partially observable Markov decision process abstraction when $h_t = \emptyset$ and m_t is dropped, but is rich enough to accommodate LLM-specific artifacts: reasoning traces, retrieval snippets, tool schemas, human corrections, natural language critiques, and rejected-action classes.

Design principle of ATDP. We enumerate the design principles for ATDP below:

- *Decision-relevant bounded revelation:* ATDP should record enough information to improve behavior but should not require exposing every internal token or hidden reasoning trace. Concretely, a self-evolving agent needs access to enough structure to support credit assignment, causal diagnosis, and replay, but it does not generally need unrestricted disclosure of every hidden token or chain-of-thought fragment, which usually cannot provide auditable learning signals.
- *Unification across frameworks and tasks:* Most existing agent logs are framework-specific, and most RL datasets are task-specific. Furthermore, most enterprise telemetry is optimized for debugging, reliability, and compliance, not for online agentic RL. One core innovation of ATDP is to make the unit of learning neither a prompt-response pair nor an opaque trace, but a typed, auditable, credit-assignable event record.
- *Credit assignability:* ATDP must make the trajectory possible to answer the question of “which observation, prompt fragment, retrieval result, tool call, memory item, or guardrail decision contributed to success or failure?”, which requires storing not only chosen actions but also the key decision context in which they were chosen. For example, for tool calls, ATDP should store tool version, argument schema, permission scope, latency, error class, return object, and whether the result was later trusted, ignored, corrected, or contradicted.
- *Late-bound learning signals:* Many useful rewards and critiques arrive after the acting step, e.g., a user correction in the next conversational turn, a failing test, a later human annotation, or a slower remote evaluator. ATDP should therefore allow an event’s reward field to be updated or augmented after initial logging while preserving immutability of the original causal record. This is essential for enterprise settings in which some judgment signals are asynchronous, policy-mediated, or sampled. ATDP should treat this as a first-class property of agent learning data.
- *Versioned replayability:* Every event should be attributable not just to a model identifier, but to the exact execution envelope that produced it, including harness schema, tool version, retrieval index snapshot, guardrail configuration, and policy LLM version or checkpoint. Without these fields, “agent experience” becomes statistically useful but operationally non-reproducible.
- *Governed observability:* Enterprise agentic interaction data could include privacy, security, and lineage fields from the beginning. ATDP should support redaction status, data-classification labels, tenant identifiers, retention policy, consent or legal basis, human-review status, and training eligibility. It should also support split visibility: a production debugger may see redacted traces, while a controlled training job may access sealed, policy-approved fields.

4 Comprehensive Agentic Data Proxy

Given the proposal of ATDP, the next question to ask is:

How to capture such data for enterprise-level heterogeneous agentic workflows?

Our answer is to implement a comprehensive data proxy. Concretely, we interpret the data proxy not merely as an API gateway, a tracing exporter, or a logging service; rather, it is the essential mechanism that converts production workloads into governed learning material. We introduce the second pillar of self-evolving agents as:

Key claim (ii): *Self-evolving agents require an enterprise-grade data proxy that can intercept, capture, anonymize, persist, and replay agentic workloads across heterogeneous frameworks and model providers, with lossless serialization for multiple agentic RL paradigms.*

ATDP specifies what a learning-ready trajectory should contain, while the enterprise agentic data proxy specifies *how such trajectories are captured in production*. The proxy sits between the agent and the LLMs (either deployed internally or by an external provider), between the agent and its tools, between the agent and its short- and long-term memory systems, and between the agent and human feedback channels. The key purpose is to transform production work into governed learning data without forcing every agent team to rewrite its application within an enterprise.

Design principle of data proxy. We introduce our key design of the data proxy as follows:

- *Existing framework-agnostic interception:* Enterprises usually will not standardize on a single agent framework. For example, some teams may use LangChain [31] or LangGraph [32]; some teams will use CrewAI [33]; some teams will use the OpenAI Agents SDK [34] or Claude Agent SDK [35]; some will use MCP-connected tools [25]; others will write custom orchestration code. The data proxy must therefore intercept at stable boundaries: model API calls, tool invocations, retrieval calls, memory operations, file-system or browser actions, human approval events, and final user feedback.
- *Lossless ATDP emission:* Every intercepted call should be converted into an ATDP event stream and persisted in a form suitable for sequential learning by the data proxy. For LLM calls, lossless means preserving all fields that are eligible and necessary for learning under policy, including prompt-template fingerprint, system-prompt version, exposed tools, decoding parameters, sampled output, token IDs where available, log probabilities where available, and model version or checkpoints. For tool calls, Lossless means storing inputs, outputs, schema, version, permission scope, error class, and downstream use.
- *Replay capability:* A trajectory that cannot be replayed is not trustworthy training data. The replay capability requires capturing tool inputs and outputs, environment state, LLM versions where licensable, file or database snapshots where permitted, and external side-effect boundaries. The replay capability also requires distinguishing deterministic replay, approximate replay, and non-replayable events. For example, A monitoring trace can include statements, such as “The agent called tool X and failed.” A training proxy must enable the system to ask: “Would the agent have succeeded under a different prompt, model, memory, retrieval policy, or tool schema?” Replay is the key feature that distinguishes a learning proxy from a monitoring proxy.
- *Cross-tenant aggregation with isolation:* An enterprise may want a single coding agent to improve from trajectories across multiple product teams, but each team may have different repository permissions, data-classification rules, and legal restrictions. The data proxy should therefore support isolated tenant stores, policy-based aggregation, federated or split-learning-style training jobs where needed, and tenant-aware evaluation. This is not only a privacy requirement but also a learning requirement: without tenant metadata and isolation, the system cannot know whether an update learned from one team should generalize to another.
- *Reward harvesting:* Agentic workloads contain many weak and delayed rewards: user replies, ticket reopen rates, test failures, compiler errors, human corrections, escalation decisions, refund reversals, approval latency, downstream edits, and abandonment. Following the central observation from OpenClaw-RL [2] that every interaction can generate a next-state signal, such as a user reply, tool output, terminal state change, or GUI state change, an enterprise data proxy should generalize this observation by treating operational state changes as candidate learning signals. Not all signals are rewards, and not all rewards are safe to optimize directly, but the data proxy must capture them before the control plane can decide how to use them.
- *Data integrity before learning.* The data proxy should enforce redaction, access control, retention, and learning eligibility before the trajectory data enters its learning queues. This is the inverse of the common pattern in which logs accumulate first, and governance is attempted later. In a self-evolving system, the proxy is not merely an observability collector; it is the first safety boundary of the RL learning loop.

5 Agent Evolution Control Plane

Given the support of a comprehensive agentic data proxy, the last key question is about:

When and how to trigger corresponding RL optimization for self-evolving?

To answer this question, we propose a unified agent evolution control plane that automatically selects the appropriate intervention surface—memory, skill, harness, or weights—based on trajectory statistics, performance drift, risk, and cost. These are not auxiliary details; they are the system’s preconditions for self-evolving agents at scale. In other words, self-evolution is not a single optimizer; rather, it is a decision problem under governance with multiple intervention surfaces. We summarize the third pillar of self-evolving agents as:

Key claim (iii): *Self-evolving agents require a unified evolution mechanism that supports both model-weight updates through diverse RL algorithms and in-context harness engineering, and that automatically triggers the appropriate intervention based on trajectory statistics, performance drift, safety constraints, and cost.*

A prototype formulation. We introduce a prototype formulation of the control plane. Formally, we define a deployed self-evolving agent $\mathcal{A}$ at time t as:

$$\mathcal{A}_t = \langle \pi_{\theta_t}, \mathcal{H}_{\psi_t}, \mathcal{M}_t, \mathcal{T}_t, \mathcal{G}_t \rangle.$$

Here, π_{θ_t} is the policy LLM parameterized by θ , $\mathcal{H}_{\psi_t}$ is the in-context harness (the harness policy parameterized by ψ), $\mathcal{M}_t$ is memory, $\mathcal{T}_t$ is the tool repertoire and tool schemas, and $\mathcal{G}_t$ is the safe governance and guardrail configuration. The targeted objects of self-evolution could be diverse, e.g., including system prompt instructions, developer instructions, agentic prompt templates, tool descriptions, routers, memory retrieval policies, memory policies, planning templates, subagent definitions, skill libraries, etc.

During self-evolving, the control plane observes a window of ATDP trajectories $\mathcal{D}_t = \{\tau_i\}_{i=t-W}^t$ and selects an evolution action:

$$u^* = \arg \max_{u \in \mathcal{U}} [\mathcal{J}_{\mathcal{A}}(u \mid \mathcal{A}_t, \mathcal{D}_t)].$$

Where $\mathcal{J}_{\mathcal{A}}$ generally measures the improvement of the evolution of the agent $\mathcal{A}$. And the evolving action set $\mathcal{U}$ could include: (i) update of policy LLM π_{θ_t} (e.g., supervised fine-tuning, direct preference optimization, on-policy RL); (ii) update of in-context harness $\mathcal{H}_{\psi_t}$ (e.g., skill patch, prompt edit); (iii) update of memory $\mathcal{M}_t$ (e.g., retrieval-policy update); (iv) update of repertoire and tool schemas $\mathcal{T}_t$ (e.g., tool-description edit, tool-schema change); (v) *rollback*, and (vi) no-op, under the safe governance control $\mathcal{G}_t$.

Control plane implementation considerations. We summarize some key implementation considerations of the agent evolution control plan below:

- *Multi-surface adaptation:* The control plane should explicitly encode that different failure modes belong to different intervention classes. For example, if an agent’s trajectories show recurring missing facts or reusable procedural lessons with a narrow scope, a memory insertion is usually the cheapest and safest update. If failures cluster around tool routing, retrieval formatting, guardrail wording, or developer-message structure, then harness editing is usually more appropriate; If, by contrast, the same category of failure persists across many tenants, tasks, and tool configurations, then the evidence suggests that the problem is probably not local to memory or harness and demands a policy update using RL, process-reward learning, or distillation.
- *Automatic triggering from trajectory statistics rather than anecdotal inspection:* The control plane should operate on explicit statistics such as evaluator scores, user correction rates, process-reward estimates, tool-specific failure clusters, canary deltas, cost per successful task, and drift in workload composition. We believe these signals would be useful, e.g., OpenClaw-RL turns next-state signals into both scalar process rewards and directional hints [2]; AgentPRM shows the value of turn-wise rewards [36]; and RLAnything combines step-wise and outcome signals [37]. Thus, our essential technical consideration is to promote these statistics from monitoring artifacts into intervention triggers so that the self-evolution can be enabled by real-world agentic workflows.

- *Algorithm pluralism under a unified trigger interface*: We believe the next-generation agentic RL system should not hard-code a single particular learning paradigm—Concretely, for policy model weight update, the backend may involve on-policy or near-on-policy RL, process-reward learning, or on-policy distillation from different directive feedback; for harness-space adaptation, the agentic RL system may involve prompt optimisation, context evolution, code search, or trajectory-aware verbal editing; for memory or skill updates, the agentic RL system may involve trajectory distillation into reusable workflow or reasoning objects. The control plane should organize such diverse paradigms instead of erasing these differences — it decides which RL algorithm should be invoked, with which data slice, under which constraints, and on what deployment path.
- *Safe audited staged deployment*: Automatic triggering of agent self-evolution should not imply unreviewed hot-swapping of agent behavior. Every intervention should carry its own promotion path: shadow evaluation, retrospective replay checks, offline regression tests, canary rollout, rollback semantics, and differential monitoring against the incumbent version. This is especially important because the safety literature on personal agents already shows that persistent state, capabilities, and knowledge can create substantial attack surfaces. The right practice is not abandoning self-evolution, but ensuring evolution is stage-gated and auditable. A unified control plane is precisely the place where such gates belong.
- *Replay-first evaluation*: Before an agentic workflow update reaches users, the evolved agent should be evaluated against replayed trajectories and counterfactual variants. For example, a harness edit should be tested on past failures and known successes. A tool-schema change should be tested on tool-call replay. A policy model update should be tested on held-out trajectories, safety sets, tenant-specific evals, and distribution-shift probes. Note that this control-plane implementation also underscores why the data proxy’s replay capability is essential.
- *Versioned provenance and rollback*: We force every evolved artifact to be versioned: LLM checkpoint, prompt, tool schema, memory item, retrieval policy, guardrail, skill file, router, and dataset. Rollback should not be a manual emergency procedure; it should be a first-class action in evolving action set $\mathbf{U}$. A self-evolving agent that cannot explain what changed is not self-evolving at the enterprise level—it is merely drifting.

6 AREAL2.0: Prototype System Design and Implementation

To make the preceding architectural claims more concrete *without* claiming to realize the full landscape of the self-evolving agents substrate, we implement AREAL2.0 as an exploratory prototype focused on one specific evolution path: online policy LLM weight updates from deployed agent trajectories. Our goal is not to implement the entire self-evolving agent substrate described above, nor to cover all possible intervention surfaces such as memory insertion, skill patching, harness editing, or tool-schema evolution. Instead, AREAL2.0 uses policy LLM updating as a representative example to show how an existing RL framework can be modified from an offline post-training system into a simple online learning paradigm. Concretely, AREAL2.0 exposes the rollout and training workers of the original AREAL framework as agent service-oriented computation components, allowing the existing agent service(s) to replace its standard LLM inference backend with a AREAL2.0-managed agent-compute worker. This design lets deployed agent interactions be served, captured, stored, and consumed by an online RL training loop with minimal changes to the surrounding agent harness.

AREAL2.0 core design. The core design goal of AREAL2.0 is to adapt the rollout and training workers in current RL systems as a replacable inference service backbone into any deployed online agent-service without modifying any existing agent harness implementation. In other word, we follow the core desgin:

AREAL2.0 prototype core desgin: *Instead of treating RL infrastructure as an offline post-training pipeline that is disconnected from online deployed agent workloads, AREAL2.0 exposes the same computation units (e.g., GPU workers) used for rollout and policy training as micro-service components that can be plugged into any existing online agent deployment.*

Concretely, AREAL2.0 introduces an elegant lightweight extension to the current AREAL framework, where the original rollout and training workers are wrapped behind an agentic micro-service abstraction, allowing

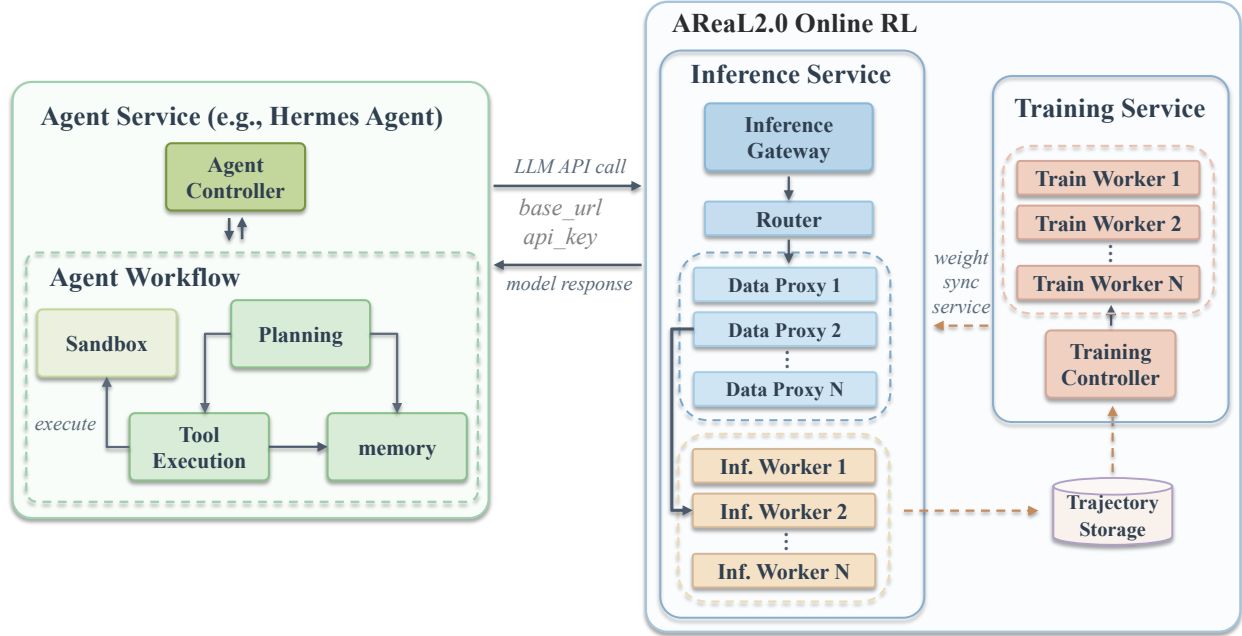


Figure 1 AREAL2.0 online RL workflow illustration: An existing agent service keeps its original planning, tool execution, sandbox, and memory modules, while redirecting LLM API calls to the gateway, then the router routes requests through data proxies to agent-compute workers, records deployed trajectories, and connects inference serving with online RL training for policy-model weight updates.

any existing agent service to replace its standard LLM inference backend (e.g., local deployment) with an AREAL2.0-managed agent-compute worker. This enables agentic trajectories generated by deployed agents interactions to be captured, stored, and consumed by the online RL training loop without requiring the agent application itself to be rewritten around a different offline RL runtime.

AREAL2.0 implementation. Our prototype system realizes this abstraction through four core components: a *gateway*, a *router*, a *data proxy*, and an *agent-compute worker*. Together, these components decouple public service protocols, session placement, trajectory management, and model inference or training computation (See Figure 1). This separation is essential for agentic RL workloads, where requests are often multi-turn, tool-augmented, and stateful, while training requires structured trajectories with token-level metadata and reward signals.

- *Gateway.* The gateway is the public entry point of the agent-service stack in the online RL system. The core functionality of the gateway is to expose AREAL2.0 as a replacement of the standard inference backend to an existing deployable agent service. From the perspective of the application, replacing an ordinary inference endpoint, e.g., an SGLang or vLLM server, with the AREAL2.0 gateway is sufficient to redirect LLM calls into the online RL-enabled runtime. Internally, the gateway normalizes external LLM requests, authenticates access, and forwards agent turns into the online RL service fabric. This gateway essentially minimizes integration cost: the agent service continues to communicate through regular inference-style APIs, while AREAL2.0 gains visibility into the interaction stream needed for online RL.
- *Router.* The router provides lightweight session-affinity management for multiple online RL training jobs. Note that agentic RL workloads are naturally stateful: a single task may involve multiple turns, tool calls, intermediate observations, and delayed rewards. The router assigns each session to a data proxy and preserves this assignment across turns. This allows AREAL2.0 to support many concurrent agentic RL workloads while maintaining consistency of session state. Concretely, the router manages placement and affinity metadata, leaving high-volume interaction traffic to the data proxy and worker layers.
- *Data Proxy.* The data proxy implements the control-plane management for agentic data traffic and storage.

The data proxy mediates between external service requests and the compute worker, records agentic data trajectories (e.g., conversation histories, tool-call events and responses), and prepares trajectory data for downstream training. In this sense, the data proxy is the component that converts ordinary deployed-agent traffic into online RL-consumable experience. The data proxy ensures that agent interactions are not merely served to users but are also structured as training data with the information needed for policy improvement.

- *Agent-Compute Worker.* The agent-compute worker is the execution abstraction that connects deployed agent services to the rollout and training backends in AREAL2.0. The core function is to wrap the various inference rollout engines, such as SGLang and vLLM, together with training workers such as Megatron- or FSDP-based actors, behind a micro-service interface. This worker abstraction allows compute resources to be dynamically allocated and deallocated according to the trajectory stream and training demand. As a result, AREAL2.0 can serve online agent requests, collect trajectories, and update the underlying policy within a unified service-oriented architecture.

Motivating example: online RL for hermes agent by AREAL2.0. We implement a representative use case, i.e., supporting the online RL training for the Hermes agent service [38]. In a conventional deployment, Hermes invokes an inference backend, e.g., an SGLang worker, to obtain model responses during agent execution. With AREAL2.0, this backend can be replaced by an AREAL2.0-managed agent-compute worker exposed through the gateway. The surrounding agent service remains mostly unchanged: the agent service continues to issue LLM inference requests as before, while AREAL2.0 intercepts the interaction stream, records trajectories, and connects them to the online RL training loop. This example illustrates the central benefit of the AREAL2.0 design. Rather than building a separate RL environment that attempts to imitate production behavior, the system reuses data generated by the original agent service itself. The agent’s native workflow, including multi-turn interaction and tool use, becomes the source of agentic data protocol trajectories. This reduces the gap between offline post-training data and deployed agent behavior, and provides a practical path toward continuously improving agents from their own service traffic.

Toward the complete self-evolving agent system substrate. The current AREAL2.0 prototype should be interpreted as a deliberately scoped proof of feasibility for online policy-model adaptation, rather than as a complete implementation of the self-evolving agent substrate envisioned by the complete lanscope of the self-evolving agent system substrate. We demonstrate one important integration principle: existing RL infrastructure can be lightly reorganized so that rollout generation, inference serving, trajectory collection, and policy optimization are connected to deployed agent workflow with minimal change of the originl online deployment. However, this prototype only covers the model-weight-update branch of the broader control-plane action space. A full self-evolving agent system still requires several capabilities beyond the current prototype, including a complete ATDP implementation with step-level decision context and governance metadata, a comprehensive data proxy that captures tool, retrieval, memory, file, browser, human-feedback, and delayed-reward events, replay and counterfactual evaluation support, tenant-aware privacy and training-eligibility enforcement, and an evolution control plane that can automatically choose among memory updates, skill patches, harness edits, tool-schema changes, policy updates, rollback, or no-op. Thus, AREAL2.0 should be viewed as an initial system step that grounds the policy-update case, while the full multi-surface, governed, replayable, and automatically triggered self-evolution loop remains the larger research and systems agenda.

7 Conclusion

We argued that self-evolving agents can be enabled by the next generation of agentic RL systems. Our key position is that the foremost bottleneck for enterprise-scale self-evolving agents is not only the absence of more powerful LLMs or more effective RL algorithms, but the absence of a system substrate that can transform deployed agent experience into governed, credit-assignable, and replayable learning material. To make this possible, we proposed three co-designed pillars: a standardized agent trajectory data protocol, an enterprise-grade agentic data proxy, and a unified agent evolution control plane. The *first pillar*, the agent trajectory data protocol (ATDP), reframes agent experience as typed, step-level, RL-grade event data rather than opaque logs or prompt-response pairs. Such a protocol should preserve the decision context, action, outcome, reward signal, metadata, provenance, and governance status needed for credit assignment and replay. The *second pillar*, the comprehensive agentic data proxy, specifies how heterogeneous production

interactions across models, tools, retrieval systems, memory stores, and human-feedback channels can be intercepted, redacted, persisted, annotated, and replayed as learning-ready trajectories. The *third pillar*, the agent evolution control plane, determines when and how the agent should evolve by inserting memory, patching skills, editing a harness, changing a tool schema, updating policy LLM weights through RL, rolling back, or doing nothing. Together, these three pillars shift self-evolving agents from an algorithmic aspiration to an enterprise systems problem. As a concrete initial step, we implement AREAL2.0 that demonstrates the policy model update branch of this agenda by connecting deployed online agent serving, trajectory collection, inference workers, and RL training workers within one online learning workflow—the full self-evolving agent substrate still requires complete ATDP support, governed replay, and automatic multi-surface evolution beyond policy-model weight updates.

References

- [1] OpenClaw. Openclaw: The ai that actually does things, 2026.
- [2] Yinjie Wang, Xuyang Chen, Xiaolong Jin, Mengdi Wang, and Ling Yang. Openclaw-rl: Train any agent simply by talking. [arXiv preprint arXiv:2603.10165](#), 2026.
- [3] Peng Xia, Jianwen Chen, Xinyu Yang, Haoqin Tu, Jiaqi Liu, Kaiwen Xiong, Siwei Han, Shi Qiu, Haonian Ji, Yuyin Zhou, et al. Metaclaw: Just talk—an agent that meta-learns and evolves in the wild. [arXiv preprint arXiv:2603.17187](#), 2026.
- [4] Ziyu Ma, Shidong Yang, Yuxiang Ji, Xucong Wang, Yong Wang, Yiming Hu, Tongwen Huang, and Xiangxiang Chu. Skillclaw: Let skills evolve collectively with agentic evolver. [arXiv preprint arXiv:2604.08377](#), 2026.
- [5] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. [Advances in neural information processing systems](#), 36:8634–8652, 2023.
- [6] Huichi Zhou, Siyuan Guo, Anjie Liu, Zhongwei Yu, Ziqin Gong, Bowen Zhao, Zhixun Chen, Menglong Zhang, Yihang Chen, Jinsong Li, et al. Memento-skills: Let agents design agents. [arXiv preprint arXiv:2603.18743](#), 2026.
- [7] Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. 2026.
- [8] Wei Fu, Jiaxuan Gao, Xujie Shen, Chen Zhu, Zhiyu Mei, Chuyi He, Shusheng Xu, Guo Wei, Jun Mei, WANG JIASHU, et al. Areal: A large-scale asynchronous reinforcement learning system for language reasoning. In [The Thirty-ninth Annual Conference on Neural Information Processing Systems](#), 2025.
- [9] Kaiyan Zhang, Yuxin Zuo, Bingxiang He, Youbang Sun, Runze Liu, Che Jiang, Yuchen Fan, Kai Tian, Guoli Jia, Pengfei Li, et al. A survey of reinforcement learning for large reasoning models. [arXiv preprint arXiv:2509.08827](#), 2025.
- [10] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. [Advances in neural information processing systems](#), 35:27730–27744, 2022.
- [11] Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, et al. Constitutional ai: Harmlessness from ai feedback. [arXiv preprint arXiv:2212.08073](#), 2022.
- [12] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. [arXiv preprint arXiv:1707.06347](#), 2017.
- [13] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. [Advances in neural information processing systems](#), 36:53728–53741, 2023.
- [14] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. [arXiv preprint arXiv:2501.12948](#), 2025.
- [15] Guibin Zhang, Hejia Geng, Xiaohang Yu, Zhenfei Yin, Zaibin Zhang, Zelin Tan, Heng Zhou, Zhongzhi Li, Xiangyuan Xue, Yijiang Li, et al. The landscape of agentic reinforcement learning for llms: A survey. [arXiv preprint arXiv:2509.02547](#), 2025.

- [16] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. [arXiv preprint arXiv:2210.03629](#), 2022.
- [17] Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic "differentiation" via text. [arXiv preprint arXiv:2406.07496](#), 2024.
- [18] Jiaxuan Gao, Wei Fu, Minyang Xie, Shusheng Xu, Chuyi He, Zhiyu Mei, Banghua Zhu, and Yi Wu. Unlocking long-horizon agentic search with large-scale end-to-end rl. In [The Fourteenth International Conference on Learning Representations](#), 2026.
- [19] Zhiyu Mei, Wei Fu, Kaiwei Li, Guangju Wang, Huanchen Zhang, and Yi Wu. Real: Efficient rlhf training of large language models with parameter reallocation. [arXiv preprint arXiv:2406.14088](#), 2024.
- [20] Yinmin Zhong, Zili Zhang, Bingyang Wu, Shengyu Liu, Yukun Chen, Changyi Wan, Hanpeng Hu, Lei Xia, Ranchen Ming, Yibo Zhu, et al. Optimizing {RLHF} training for large language models with stage fusion. In [22nd USENIX Symposium on Networked Systems Design and Implementation \(NSDI 25\)](#), pages 489–503, 2025.
- [21] Junyu Wu, Weiming Chang, Xiaotao Liu, Guanyou He, Haoqiang Hong, Boqi Liu, Hongtao Tian, Tao Yang, Yunsheng Shi, Feng Lin, et al. G-core: A simple, scalable and balanced rlhf trainer. [arXiv preprint arXiv:2507.22789](#), 2025.
- [22] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework. In [Proceedings of the Twentieth European Conference on Computer Systems](#), pages 1279–1297, 2025.
- [23] Yinmin Zhong, Zili Zhang, Xiaoni Song, Hanpeng Hu, Chao Jin, Bingyang Wu, Nuo Chen, Yukun Chen, Yu Zhou, Changyi Wan, et al. Streamrl: Scalable, heterogeneous, and elastic rl for llms with disaggregated stream generation. [arXiv preprint arXiv:2504.15930](#), 2025.
- [24] Zhenyu Han, Ansheng You, Haibo Wang, Kui Luo, Guang Yang, Wenqi Shi, Menglong Chen, Sicheng Zhang, Zeshun Lan, Chunshi Deng, et al. Asyncflow: An asynchronous streaming rl framework for efficient llm post-training. [arXiv preprint arXiv:2507.01663](#), 2025.
- [25] Anthropic. Introducing the Model Context Protocol. <https://www.anthropic.com/news/model-context-protocol>, 2024.
- [26] Google. Agent2agent (a2a) protocol. <https://a2a-protocol.org/>, 2025.
- [27] Yingxuan Yang, Huacan Chai, Yuanyi Song, Siyuan Qi, Muning Wen, Ning Li, Junwei Liao, Haoyi Hu, Jianghao Lin, Gaowei Chang, et al. A survey of ai agent protocols. [arXiv preprint arXiv:2504.16736](#), 2025.
- [28] Justin Fu, Aviral Kumar, Ofir Nachum, George Tucker, and Sergey Levine. D4rl: Datasets for deep data-driven reinforcement learning. [arXiv preprint arXiv:2004.07219](#), 2020.
- [29] Sabela Ramos, Sertan Girgin, Léonard Hussenot, Damien Vincent, Hanna Yakubovich, Daniel Toyama, Anita Gergely, Piotr Stanczyk, Raphael Marinier, Jeremiah Harmsen, et al. Rlds: an ecosystem to generate, share and use datasets in reinforcement learning. [arXiv preprint arXiv:2111.02767](#), 2021.
- [30] Yueqi Song, Ketan Ramaneti, Zaid Sheikh, Ziru Chen, Boyu Gou, Tianbao Xie, Yiheng Xu, Danyang Zhang, Apurva Gandhi, Fan Yang, et al. Agent data protocol: Unifying datasets for diverse, effective fine-tuning of llm agents. [arXiv preprint arXiv:2510.24702](#), 2025.
- [31] LangChain, Inc. LangChain: The agent engineering platform. <https://github.com/langchain-ai/langchain>, 2025.
- [32] LangChain, Inc. LangGraph: Build resilient language agents as graphs. <https://github.com/langchain-ai/langgraph>, 2025.
- [33] crewAI, Inc. CrewAI: Framework for orchestrating role-playing, autonomous AI agents. <https://github.com/crewAIInc/crewAI>, 2025.
- [34] OpenAI. OpenAI Agents SDK: A lightweight, powerful framework for multi-agent workflows. <https://github.com/openai/openai-agents-python>, 2025.
- [35] Anthropic. Claude Agent SDK. <https://github.com/anthropics/claude-agent-sdk-python>, 2025.

- [36] Zhiheng Xi, Chenyang Liao, Guanyu Li, Zhihao Zhang, Wenxiang Chen, Binghai Wang, Senjie Jin, Yuhao Zhou, Jian Guan, Wei Wu, et al. Agentprm: Process reward models for llm agents via step-wise promise and progress. In Proceedings of the ACM Web Conference 2026, pages 4184–4195, 2026.
- [37] Yinjie Wang, Tianbao Xie, Ke Shen, Mengdi Wang, and Ling Yang. Rlanything: Forge environment, policy, and reward model in completely dynamic rl system. arXiv preprint arXiv:2602.02488, 2026.
- [38] Nous Research. Hermes agent: The self-improving ai agent built by nous research. <https://github.com/NousResearch/hermes-agent>, 2026. Accessed: 2026-06-30.