Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Toward Zero Fixed Code: Complete-Generation-Information-Driven Self-Learning for Autonomous Robots via Multi-Layer Code Replaceability and Safe Rollback

Hong Su

suguest@126.com

Chengdu University of Information Technology

Research Article

Keywords: Autonomous robots, Zero Fixed Code, complete code replaceability, large language models

Posted Date: April 9th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-9356516/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

Toward Zero Fixed Code: Complete-Generation-Information-Driven Self-Learning for Autonomous Robots via Multi-Layer Code Replaceability and Safe Rollback

Hong Su

***Abstract*—Autonomous robots operating in complex real-world environments require the ability to continuously improve their behavior beyond what can be fully specified by predefined code. However, conventional robotic systems usually rely on fixed hand-crafted logic, at least for some core components, which makes them difficult to adapt when new task constraints, environmental changes, or previously unseen problems arise after deployment. In this paper, we propose a complete-generation-information-driven self-learning framework for autonomous robots, in which the robot preserves not only the currently executable code but also the full information chain that produced it, including generation prompts, task objectives, execution traces, environmental feedback, and historical logs. Based on this preserved information, the robot can interact with a large language model to revise its own executable logic when new requirements appear. The proposed framework further supports multi-layer code replaceability, allowing not only task-solving code but also higher-level operational logic, such as update policy and scheduler logic, to be replaced during runtime. To maintain survivability during self-evolution, we also introduce a redundant replacement and rollback mechanism that enables the system to reject faulty candidate code and recover a previously valid version without interrupting continued operation. Experimental results show that on an 11-episode progressively changing task benchmark, the proposed method achieved a final cumulative success rate of 1.0, compared with 0.2727 for the fixed-code baseline, while completing 7 update events, 11 successful layer replacement events, and 1 rollback event.**

***Index Terms*—Autonomous robots, Zero Fixed Code, complete code replaceability, large language models**

I. Introduction

Recent advances in large language models (LLMs) [1] have greatly strengthened the ability of artificial intelligence systems to generate executable code, revise programs, and solve complex problems through natural language interaction [2]. This development suggests a new possibility for autonomous robotics: instead of relying on manually predefined logic as the long-term foundation of the system, a robot may gradually grow its own executable behavior through repeated interaction with the environment and continual support from increasingly capable LLMs.

H. Su is with the School of Computer Science, Chengdu University of Information Technology, Chengdu, China.
E-mail: suguest@126.com.

For autonomous robots deployed in real environments, one fundamental limitation of conventional design is that at least part of the system logic is usually fixed in advance [3]. Even when machine learning is used, key components such as business logic, scheduling logic, update logic, or safety handling mechanisms are often predefined by human designers and remain unchanged during runtime. This design is practical for narrow and stable scenarios, but it becomes increasingly restrictive when the robot faces new task constraints, unforeseen environmental changes, or long-term deployment conditions that cannot be fully anticipated beforehand.

A more ambitious question is therefore whether a robot can move toward *zero fixed code*, that is, whether all non-root executable logic can in principle be replaced and improved during operation. Such a capability would allow the robot not only to solve current tasks, but also to continuously reconstruct its own logic as it encounters new requirements. In this sense, robot intelligence would no longer be limited to selecting actions under a fixed framework; it would include the ability to grow the framework itself.

This problem becomes especially important as both the environment and the available external intelligence continue to evolve. Real-world environments are not static: new obstacles, new task rules, new safety requirements, and new combinations of constraints may all emerge over time. At the same time, LLMs themselves are also improving, which means a deployed robot may later have access to stronger code-generation and reasoning capabilities than it had at the time of initial deployment. A robot architecture that can preserve sufficient information about its previous code generation and execution process may therefore continue to benefit from future improvements in LLM capability, rather than remaining permanently bounded by its initial implementation.

However, moving toward zero fixed code is difficult for at least three reasons. First, replacing code safely is much harder than adjusting parameters within a fixed model. If newly generated logic is directly deployed without protection, a faulty update may destroy the robot's ability to continue operating or to perform future self-improvement. Second, learning from runtime experience requires more than storing the latest code version. To revise logic effectively, the system should preserve the complete information chain behind previous generations, including why the code was generated, what task it targeted, how it executed, what feedback it received,

and how earlier revisions succeeded or failed. Third, if self-evolution is to extend beyond one task-specific module, the architecture must support replacement across multiple code layers while maintaining operational continuity.

Existing work on LLM-based robotics has shown that LLMs can generate plans, code, and reward-related logic for robotic systems [4, 5, 6]. Other recent studies have further shown that robot programs can be improved through execution-grounded rewriting or critic-based correction [7, 8]. Meanwhile, self-improving agent research has begun to explore recursive code modification and open-ended agent evolution [9, 10]. These advances strongly motivate the possibility of self-growing robotic systems. Nevertheless, existing studies have not sufficiently addressed a unified autonomous robot framework that simultaneously preserves complete generation information, supports multi-layer runtime code replaceability, and maintains safe rollback during self-evolution.

To address this gap, this paper proposes a complete-generation-information-driven self-learning framework for autonomous robots. The key idea is that the robot preserves not only its currently executable code, but also the full generation chain that produced it, including prompts, goals, generated logic, execution traces, environmental feedback, and historical logs. Based on this preserved information, the robot can interact with an LLM to revise its own executable logic when new requirements appear. Furthermore, the framework is designed for multi-layer code replaceability, so that not only task-solving logic but also higher-level operational logic, such as update policy and scheduler logic, can be replaced during runtime. To ensure survivability, the framework also introduces redundant replacement and rollback mechanisms that allow the system to reject faulty candidate code and recover a previous valid version without interrupting continued operation.

The main contributions of this paper are as follows:

- We propose a complete-generation-information-driven self-learning framework for autonomous robots, in which the robot preserves the full generation-and-execution chain of its executable logic rather than only the latest code.
- We introduce a multi-layer replaceability mechanism that allows runtime replacement not only of task-solving code but also of higher-level operational code, moving toward the goal of zero fixed code for non-root logic.
- We design a safe replacement architecture with rollback support, enabling the robot to maintain continued operation even when a candidate replacement fails.

The remainder of this paper is organized as follows. Section II reviews related work. Section III presents the proposed model. Section IV describes the verification setting and experimental results. Finally, the paper concludes with discussion and future directions.

II. Related Work

Recent research has increasingly explored the use of large language models (LLMs) and broader foundation models in robotics, especially for task planning, action generation, reward design, and embodied decision-making. Existing surveys show that LLM-driven robotics has rapidly expanded from language-conditioned planning to more integrated robotic systems that combine reasoning, perception, and execution [6].

However, most existing work is centered on using LLMs to produce plans, code, or rewards for specific tasks, or on improving task performance through feedback. Comparatively less attention has been paid to autonomous robots that preserve the full generation chain of their executable logic, support runtime replacement across multiple code layers, and maintain safe rollback during self-evolution.

In this section, we review related work from three aspects: 1) LLM-based robotic planning and code generation, 2) execution-grounded refinement of robot programs, and 3) self-improving agents and safe runtime code evolution.

A. LLM-Based Robotic Planning and Code Generation

One major research direction uses LLMs as high-level reasoning engines for robotics, where natural-language instructions are translated into robot plans, executable programs, or other structured control representations. A representative example is ProgPrompt, which proposes a program-like prompting structure that constrains LLM outputs using situated robot actions and environment objects, enabling executable task plans in both simulated household environments and real robotic manipulation settings [4].

Another related line uses coding-oriented LLMs to generate control-related logic for robotics and embodied learning. Eureka, for example, employs an LLM to generate reward code and iteratively improve it for robot skill learning, showing that code-generating LLMs can contribute not only to symbolic planning but also to lower-level learning-related components such as reward functions [5].

More broadly, recent surveys on LLMs for robotics indicate that these models are being applied across planning, perception, control, interaction, and embodied reasoning [6]. These reviews also point out important limitations in grounding, robustness, safety, and long-horizon reliability in real robotic deployments.

Despite these advances, most existing work in this category treats the LLM primarily as a generator of task plans, reward logic, or execution code for a given problem instance. In contrast, our work focuses on how an autonomous robot can preserve the complete generation information behind its executable logic and use that preserved information to support continued self-learning and runtime code replacement beyond a single task-generation episode.

B. Execution-Grounded Refinement of Robot Programs

A second important research direction goes beyond one-shot plan or code generation and studies how robot programs can be improved using execution feedback, environmental observations, or verification signals. In this line of work, the LLM is not used only once to produce an initial plan; instead, generated robot logic is iteratively revised according to what happens during execution.

A recent representative example is Act–Observe–Rewrite (AOR), which treats executable Python controller code as the object of reasoning and improvement. In AOR, a multimodal

LLM observes execution outcomes and visual evidence, then rewrites robot manipulation code between trials to address observed failure modes [7]. This is closely related to our motivation in that robot behavior is revised through code rather than only through fixed symbolic plans or parameter tuning.

Another recent example is RoboCritics, which augments LLM-based robot programming with external robotics-informed critics that inspect execution traces and provide structured feedback about issues such as unsafe motions, collisions, and other motion-level violations [8]. This work highlights the importance of execution-grounded analysis and iterative correction for improving the reliability of generated robot programs.

More generally, recent work on robotic planning and execution with LLMs has increasingly adopted closed-loop or feedback-driven designs, where generated actions or programs are checked against execution results and then refined accordingly [**? ?**]. These approaches help reduce the gap between language-level generation and real robotic behavior by incorporating observations from runtime interaction.

Despite these advances, most existing execution-grounded refinement methods are still centered on improving the correctness or safety of a particular robot program for a current task or task family. In contrast, our work treats execution-grounded improvement as only one part of a broader self-evolution process. The key difference is that we preserve the complete generation information behind each executable logic instance, including prompts, task goals, code, runtime traces, feedback, and logs, and use this information not only to revise task-solving code but also to support multi-layer runtime code replacement and rollback-safe continued evolution.

C. Self-Improving Agents and Safe Runtime Code Evolution

A third closely related research direction studies agents that can improve themselves by modifying their own logic, workflows, or code. Compared with task-level robot planning, this line of work is more directly related to recursive self-improvement, because the object being optimized is no longer only the current task solution, but also the agent's own future problem-solving process.

A representative example is Gödel Agent, which proposes a self-referential agent framework inspired by the classical Gödel machine. Instead of relying on fixed optimization routines, Gödel Agent uses LLM-based prompting to recursively modify its own logic and behavior under high-level objectives, demonstrating continuous self-improvement on reasoning and agent tasks [9]. This work is highly relevant because it explicitly challenges the assumption that the improvement mechanism itself must remain fixed.

Another important recent example is Darwin Gödel Machine (DGM), which studies open-ended self-improvement through repeated code modification and empirical validation [11]. DGM maintains an archive of self-modified coding agents and explores multiple branches of improvement, showing that agents can improve not only task performance but also their own ability to generate further improvements. This is closely aligned with the broader idea of self-evolving systems.

Relatedly, recent work on self-improving coding agents has shown that an agent equipped with coding tools can edit its own codebase and improve benchmark performance through autonomous modification and validation [12]. At a broader level, recent surveys of self-evolving agents also show that evolution may target multiple components, including memory, tools, workflows, architecture, and adaptation strategies [10]. These findings support the view that self-improvement should not be restricted to a single fixed module.

Despite these important advances, most existing self-improving-agent work is not centered on autonomous robots operating under evolving embodied task constraints. Moreover, while self-modification is increasingly studied, safe runtime evolution remains a critical issue. In particular, for real autonomous systems, it is not sufficient to modify code successfully once; the system must also preserve continued operation when candidate modifications are faulty, and higher-level operational code should remain replaceable without destroying future upgradability. In contrast, our work focuses on an autonomous robotic setting and emphasizes three properties jointly: preservation of complete generation information, multi-layer runtime code replaceability, and rollback-safe continued evolution.

III. Model

A. Design Goal and Core Definition

The goal of the proposed model is to construct an autonomous robotic system whose task-solving logic is not constrained by fixed, manually predefined business code, but can instead be continuously generated, revised, evaluated, and replaced through large language model (LLM) interaction and environmental feedback. Unlike conventional robotic systems, where most task logic is designed in advance and only a limited set of parameters can be updated during operation, the proposed model aims to support *open-ended self-growth* under long-term deployment.

The central idea is that the robot should preserve not only the currently executable code, but also the *complete generation information* that led to that code. This information enables the system to understand how a behavior was produced, why it succeeded or failed, and how it can be improved in future iterations. In this way, the robot does not merely execute code; it continuously learns how to regenerate and refine its own logic.

Formally, let the robot state at time step t be denoted by

$$\mathcal{S}_t = (C_t, \mathcal{M}_t, \mathcal{H}_t, \mathcal{E}_t)\,, \tag{1}$$

where C_t denotes the currently active executable code, $\mathcal{M}_t$ denotes the meta-information for code generation and revision, $\mathcal{H}_t$ denotes the historical records accumulated so far, and $\mathcal{E}_t$ denotes the current environmental observations and runtime feedback.

Among them, the key object is the complete generation information, denoted by

$$\mathcal{G}_t = (P_t,\ T_t,\ C_t,\ R_t,\ F_t,\ L_t)\,, \tag{2}$$

where P_t is the prompt or generation instruction provided to the LLM, T_t is the target or intended task objective, C_t

is the generated code, R_t is the execution trace and runtime response, F_t is the feedback from the environment, and L_t is the associated log and reflection information. This definition emphasizes that the learning object is not a single output code fragment, but the entire causal chain from intention to execution outcome.

Based on $\mathcal{G}_t$, the robot performs a self-improvement mapping

$$\Phi : \mathcal{G}_{1:t} \rightarrow C_{t+1}, \tag{3}$$

where $\mathcal{G}_{1:t}$ represents the accumulated complete generation information from time step 1 to t, and C_{t+1} is the newly generated or revised code for the next stage. The mapping Φ may be implemented through one or multiple LLM interactions together with local checking, reflection, and validation procedures.

The core design objective is therefore not to optimize a single fixed policy, but to maximize the robot's long-term capability growth. Let $U(C_t, E_t)$ denote the utility of the current code under environment E_t, which may reflect task completion quality, robustness, recoverability, or adaptability. Then the objective can be expressed as

$$\max \sum_{t=1}^{T} U(C_t, E_t), \tag{4}$$

subject to the constraint that the system must preserve the minimal ability to continue operation, record information, and perform further upgrades. This means the robot should never lose the basic capability required for future self-improvement.

To characterize this requirement, we define a minimal survivability condition:

$$\Psi(C_t) = 1, \tag{5}$$

where $\Psi(C_t)$ indicates whether the currently deployed code still preserves the essential self-maintenance capabilities, such as heartbeat scheduling, information recording, environment interaction, and the ability to reconnect to LLM-based or other upgrading mechanisms. Any upgrade that violates $\Psi(C_t) = 1$ should not be accepted as the sole active implementation.

The importance of preserving complete generation information can be stated by the following proposition.

Proposition 1. Suppose two autonomous systems produce the same current executable code C_t, but one system preserves the corresponding complete generation information $\mathcal{G}_{1:t}$ while the other preserves only C_t. Then, under non-stationary environments, the former system has a strictly richer basis for future revision and fault recovery.

Proof sketch. Future revision depends not only on the current code itself, but also on information explaining the code's original target, generation rationale, runtime behavior, and past failure cases. If only C_t is preserved, then multiple distinct generation paths may correspond to the same code, and this ambiguity makes targeted correction more difficult. By contrast, when $\mathcal{G}_{1:t}$ is preserved, the system can condition future revision on prior prompts, execution traces, and environmental feedback, thereby reducing uncertainty in identifying which components should be modified. Therefore, the preserved information set of the former system strictly contains that of the latter, yielding a stronger foundation for adaptation and recovery. □

In essence, the proposed model treats autonomous learning as a process of continuously reconstructing and improving the robot's executable logic from complete historical evidence, rather than merely tuning parameters inside a fixed predefined framework. This design is especially suitable for complex real-world environments, where optimal logic cannot be fully specified in advance and where long-term adaptation is more important than one-time manual design.

B. Hierarchical Architecture

To enable continuous self-improvement without collapsing the robot's operational continuity, the proposed model adopts a hierarchical architecture with asymmetric replaceability. The key principle is that higher layers provide persistence and coordination, while lower layers provide task realization and can be more aggressively modified. This design allows the robot to keep running even when some generated logic fails, while still preserving broad freedom for self-revision.

The overall architecture is composed of three conceptual layers:

$$\mathcal{A} = \left(\mathcal{A}^{(0)},\ \mathcal{A}^{(1)},\ \mathcal{A}^{(2)}\right), \tag{6}$$

where $\mathcal{A}^{(0)}$ denotes the heartbeat layer, $\mathcal{A}^{(1)}$ denotes the cognitive adaptation layer, and $\mathcal{A}^{(2)}$ denotes the task realization layer.

The heartbeat layer $\mathcal{A}^{(0)}$ is the top persistence layer. Its responsibility is not to encode complex task logic, but to maintain the existence of an ongoing operational cycle. More specifically, it is responsible for periodically triggering execution, learning, checking, and recovery processes, and for preserving the path by which the robot can continue to interact with LLM services or other upgrading mechanisms. This layer is the only part whose *existence* must always be preserved. However, its implementation is not permanently fixed and may also be replaced, provided that a simpler or more reliable mechanism remains able to recover it if failure occurs.

The cognitive adaptation layer $\mathcal{A}^{(1)}$ performs reflection, analysis, generation, revision, comparison, and upgrade decisions. It receives complete generation information, runtime traces, historical records, and environmental feedback, and then determines whether some component of the current system should be improved. This layer therefore serves as the main bridge between raw experience and code evolution. Its output may include newly generated task logic, revised scheduling logic, updated prompts, modified checking rules, or rollback decisions.

The task realization layer $\mathcal{A}^{(2)}$ contains the concrete executable logic for specific tasks. This may include perception routines, planning procedures, control logic, reward design, policy networks, action selection modules, or domain-specific execution code. Unlike conventional systems, these components are not assumed to be permanently predefined. Instead, they are treated as revisable objects that may be regenerated from higher-layer reasoning. Thus, even the internal logic of a specific task is not a frozen hand-crafted artifact, but part of the robot's evolving structure.

The interaction among the three layers can be expressed as

$$\mathcal{A}^{(0)} \rightarrow \mathcal{A}^{(1)} \rightarrow \mathcal{A}^{(2)}, \tag{7}$$

where the arrow denotes invocation or regulation rather than one-way information flow. In practice, $\mathcal{A}^{(2)}$ produces execution results and environmental effects, these outcomes are summarized into complete generation information and fed upward to $\mathcal{A}^{(1)}$, and $\mathcal{A}^{(1)}$ then proposes revisions that are scheduled or guarded by $\mathcal{A}^{(0)}$. Therefore, the architecture is hierarchically organized in control, but cyclic in learning.

To formalize this dependency, let $X_t^{(i)}$ denote the state of layer $\mathcal{A}^{(i)}$ at time t. Then the evolution of the whole architecture may be written as

$$X_{t+1}^{(2)} = f^{(2)}\left(X_t^{(2)}, X_t^{(1)}, E_t\right), \tag{8}$$

$$X_{t+1}^{(1)} = f^{(1)}\left(X_t^{(1)}, \mathcal{G}_t, X_t^{(2)}\right), \tag{9}$$

$$X_{t+1}^{(0)} = f^{(0)}\left(X_t^{(0)}, \sigma_t, \rho_t\right), \tag{10}$$

where E_t denotes environmental observations, $\mathcal{G}_t$ denotes complete generation information, σ_t denotes system health signals such as heartbeat flags or process status, and ρ_t denotes recovery-related observations such as timeout, crash, or missing update events. These equations emphasize that different layers evolve under different types of evidence and constraints.

A critical design constraint is that replaceability should increase as the layer goes downward. Let $\mathcal{R}(\mathcal{A}^{(i)})$ denote the allowed replacement freedom of layer $\mathcal{A}^{(i)}$. Then the architecture follows

$$\mathcal{R}\left(\mathcal{A}^{(0)}\right) \leq \mathcal{R}\left(\mathcal{A}^{(1)}\right) \leq \mathcal{R}\left(\mathcal{A}^{(2)}\right). \tag{11}$$

This means task logic can be replaced most freely, cognitive adaptation logic can also evolve but under stronger validation, and the heartbeat layer must satisfy the strongest continuity constraints.

The reason for such asymmetric replaceability is that a fully unconstrained top-layer replacement may destroy the robot's ability to continue learning at all. If the system loses scheduling, logging, or the ability to reconnect to an upgrading source, then later self-improvement becomes impossible regardless of how advanced lower-layer logic may have been. Accordingly, the architecture must ensure that the capability to *continue evolution* is protected more strongly than any particular task implementation.

This intuition can be summarized by the following proposition.

Proposition 2. If the top persistence function is entirely removed without an independently recoverable substitute, then long-term self-improvement of the system is no longer guaranteed, even if all lower-layer task logic remains available.

Proof sketch. Long-term self-improvement requires repeated cycles of execution, recording, reflection, revision, and redeployment. These cycles depend on a mechanism that continues invoking them over time. If the top persistence function disappears and no external or simpler recoverable substitute exists, then the recurrence of such cycles is not guaranteed. As a result, the system may become a one-time executor of existing logic rather than a continuously evolving agent. Therefore, preserving a persistence-enabling layer is a necessary structural condition for lifelong self-improvement.

Another important architectural issue is the avoidance of deep cascaded invocation. A poor design would allow an old version to invoke a new version, which in turn invokes another newer version, leading to deep chains of nested dependence. Such cascaded execution makes fault localization, rollback, and state consistency much more difficult. Therefore, the proposed architecture separates *version storage* from *active invocation*: multiple versions may coexist in storage, but the currently active scheduler should invoke only one designated executable path for each role at a given time. In other words, replacement should change the target of invocation, rather than append another invocation layer on top of the old one.

In summary, the hierarchical architecture provides three essential properties. First, it preserves persistent operation through a minimal heartbeat layer. Second, it supports broad self-revision through a cognitive adaptation layer that reasons over complete generation information. Third, it allows concrete task logic to evolve flexibly while preventing architectural instability from uncontrolled top-layer modification. This hierarchical organization establishes the structural basis for the self-learning and safe upgrade mechanisms introduced in the following subsections.

C. Complete-Generation-Information-Driven Self-Learning

The core learning mechanism of the proposed model is not based on preserving only the final executable program, but on preserving and reusing the *complete generation information* associated with each behavior realization and code evolution event. This design allows the robot to learn from the full causal process of generation, execution, feedback, and reflection, rather than from isolated success or failure labels alone.

At time step t, the robot maintains a complete generation record

$$\mathcal{G}_t = (P_t,\ T_t,\ C_t,\ X_t,\ Y_t,\ F_t,\ L_t), \tag{12}$$

where P_t is the prompt or generation instruction given to the LLM, T_t is the intended target or task objective, C_t is the generated code, X_t denotes the contextual inputs available before execution, Y_t denotes the execution outcome, F_t denotes the environmental and self-monitored feedback after execution, and L_t denotes logs, traces, and reflection records. Compared with a conventional code-centric design, this formulation explicitly preserves both the *generation-side context* and the *post-execution evidence*.

The role of self-learning is to transform accumulated records into improved future logic. Let

$$\mathcal{H}_t = \{\mathcal{G}_1, \mathcal{G}_2, \ldots, \mathcal{G}_t\} \tag{13}$$

denote the historical memory up to time t. The robot then performs an update process

$$C_{t+1} = \Phi(\mathcal{H}_t, E_t, O_t), \tag{14}$$

where $\Phi(\cdot)$ denotes the self-improvement operator, E_t denotes the current environment state or observation context, and O_t

denotes the current operational objective. In practice, Φ may invoke an LLM, local analysis tools, rule checking procedures, testing modules, or combinations of them. The essential property is that the next code version is conditioned on the entire available evidence chain, rather than only on the most recent output.

To make this process more explicit, the self-learning cycle can be decomposed into four stages:

$$\text{Record} \rightarrow \text{Analyze} \rightarrow \text{Revise} \rightarrow \text{Validate}. \tag{15}$$

In the *Record* stage, the system stores all key artifacts related to generation and execution. In the *Analyze* stage, the system identifies potential weaknesses, such as repeated runtime faults, poor task outcomes, unstable scheduling, insufficient environmental adaptation, or missing capabilities. In the *Revise* stage, the system produces new candidate logic, which may involve changes to task code, reasoning logic, scheduling mechanisms, prompt structure, or feedback usage. In the *Validate* stage, the candidate is checked by execution results, internal constraints, redundancy comparison, or rollback rules before wider acceptance.

A key aspect of the proposed model is that self-learning is *component-aware*. The robot does not need to regenerate the whole system at every iteration. Instead, it may identify a specific component $c_t^{(k)}$ within the current system as the main source of limitation and only revise that part. Let the active code at time t be decomposed as

$$C_t = \{c_t^{(1)}, c_t^{(2)}, \ldots, c_t^{(n)}\}, \tag{16}$$

where each $c_t^{(k)}$ is a replaceable functional component. The system estimates an improvability score for each component:

$$I_t^{(k)} = \Gamma\left(c_t^{(k)}, \mathcal{H}_t, F_t\right), \tag{17}$$

where $\Gamma(\cdot)$ denotes an analysis function that evaluates how likely component k is responsible for current limitations or how promising its revision may be. The next revision target is then selected as

$$k^* = \arg\max_k I_t^{(k)}. \tag{18}$$

This allows the model to focus revisions on the most improvable or failure-relevant part of the current codebase.

The advantage of using complete generation information becomes especially important under ambiguous failure causes. Suppose the task outcome degrades at time t. If only the final code C_t is stored, then the cause of failure may be difficult to identify: the problem may originate from an inadequate prompt, a flawed code fragment, a hidden runtime assumption, an unexpected environmental shift, or an earlier mistaken revision. By contrast, when $\mathcal{H}_t$ is preserved, the system can compare generation intent, actual code, runtime traces, and feedback patterns jointly, which provides a much richer basis for diagnosis.

To capture this intuition, let the uncertainty of identifying the correct revision target be denoted by

$$\mathcal{U}(\Omega \mid \mathcal{I}), \tag{19}$$

where Ω denotes the true cause of the current limitation and $\mathcal{I}$ denotes the available information. Then preserving complete generation information implies

$$\mathcal{U}(\Omega \mid \mathcal{H}_t) \leq \mathcal{U}(\Omega \mid C_t), \tag{20}$$

because $\mathcal{H}_t$ contains C_t together with additional causal evidence such as prompts, logs, and feedback. Although exact uncertainty may be difficult to compute in practice, this inequality expresses the structural advantage of the proposed information-preserving approach.

D. Redundant Upgrade and Safe Replacement Mechanism

A central challenge of a self-evolving robotic system is that newly generated logic may be beneficial, neutral, or harmful. If each new version is directly deployed as the only active implementation, then an incorrect revision may destroy task performance, interrupt operation, or even remove the robot's future ability to continue learning. For this reason, the proposed model adopts a *redundant upgrade and safe replacement mechanism*, in which old and new versions coexist during a transition period and replacement is determined by evidence rather than immediate trust.

Let C_t^{old} denote the currently trusted implementation at time t, and let C_t^{new} denote a newly generated candidate. Instead of immediately discarding the old version, the system maintains a temporary coexistence state

$$\mathcal{V}_t = \left(C_t^{old}, C_t^{new}\right), \tag{21}$$

where both implementations remain available for comparison, fallback, and staged acceptance. The goal is not merely to test whether the new code can run once, but to determine whether it is sufficiently reliable for continued deployment under real operating conditions.

The candidate generation process can be described as

$$C_t^{new} = \Xi(\mathcal{H}_t, O_t, E_t), \tag{22}$$

where $\Xi(\cdot)$ denotes the code generation and revision procedure based on historical records $\mathcal{H}_t$, current objectives O_t, and environmental state or observations E_t. In practice, Ξ may involve LLM code generation, patch generation, script replacement plans, validation commands, file operations, or configuration revision instructions.

After generation, the system should not simply overwrite the prior version. Instead, the candidate enters an evaluation phase in which the old and new implementations are compared. Let the observed utility of version $v \in \{old, new\}$ over an evaluation window Δ_t be denoted by

$$J_t^{(v)} = \sum_{\tau \in \Delta_t} \omega_1 Q_\tau^{(v)} + \omega_2 S_\tau^{(v)} + \omega_3 A_\tau^{(v)} - \omega_4 R_\tau^{(v)}, \tag{23}$$

where $Q_\tau^{(v)}$ denotes task quality, $S_\tau^{(v)}$ denotes operational stability, $A_\tau^{(v)}$ denotes environmental adaptability, $R_\tau^{(v)}$ denotes risk or failure cost, and $\omega_1, \omega_2, \omega_3, \omega_4 \geq 0$ are weighting factors. This formulation emphasizes that acceptance should depend not only on immediate task success, but also on robustness and survivability.

A candidate version is accepted only if it satisfies both performance and survivability requirements. Formally, define the acceptance function

$$\mathcal{A}_{acc}(C_t^{new}) = \mathbf{1}\left[J_t^{(new)} > J_t^{(old)} + \epsilon \ \wedge \ \Psi(C_t^{new}) = 1\right], \tag{24}$$

where $\epsilon \geq 0$ is a safety margin and $\Psi(\cdot)$ is the survivability condition defined earlier. Thus, even if the new version performs better on a short-term metric, it should not become the sole active implementation if it loses essential capabilities such as heartbeat scheduling, logging, recovery signaling, or the ability to reconnect to upgrading mechanisms.

The coexistence mechanism is particularly important for *basic code replacement*. Recall that the system distinguishes between basic code and task-specific logic. Basic code includes functions whose loss would prevent future upgrading, such as LLM interaction, scheduling, information recording, watchdog signaling, and core execution orchestration. By contrast, task logic includes more replaceable modules such as planners, reward definitions, perception strategies, or business procedures. The proposed model allows both categories to evolve, but the replacement threshold for basic code must be stricter than that for ordinary task logic.

Let $\mathcal{B}$ denote the set of basic-code components and $\mathcal{T}$ denote the set of task-level components. Then their acceptance thresholds satisfy

$$\theta_{\mathcal{B}} \geq \theta_{\mathcal{T}}, \tag{25}$$

where $\theta_{\mathcal{B}}$ and $\theta_{\mathcal{T}}$ denote the evidence thresholds required to promote a new version into primary operation. This inequality reflects the intuitive principle that the more foundational a component is for future self-improvement, the more conservative its replacement should be.

To prevent instability during deployment, the proposed model also avoids *deep cascaded invocation*. An undesirable design would allow version 0 to invoke version 1, version 1 to invoke version 2, and so on, producing a deep chain of nested dependence. Such chains complicate control flow, blur version responsibility, and make rollback more difficult. Therefore, if a role has multiple versions available,

$$\{c^{(k,0)}, c^{(k,1)}, \ldots, c^{(k,m)}\}, \tag{26}$$

the scheduler should select only one designated active target

$$c_t^{(k,*)} = \Pi\left(\{c^{(k,0)}, c^{(k,1)}, \ldots, c^{(k,m)}\}\right), \tag{27}$$

where $\Pi(\cdot)$ denotes the current version-selection rule. The active scheduler should then invoke $c_t^{(k,*)}$ directly, rather than invoking an older version that internally calls a newer one. In this way, *version selection* is centralized, while *version execution* remains flat.

An equally important mechanism is rollback and restoration. If the new version becomes unresponsive, fails safety checks, or loses the ability to maintain heartbeat and reporting, then the system must revert to a previously trusted version. Let σ_t^{new} denote the health signal of the candidate version, such as periodic flag updates, status reports, or successful scheduler responses. A simple rollback rule may be written as

$$\text{if } \sigma_t^{new} = 0 \text{ for } \tau > \tau_{max}, \text{ then restore } C_t^{old}, \tag{28}$$

where τ_{max} is a timeout threshold. This mechanism can be enforced by internal monitoring logic or by an external hardware watchdog when the top scheduling layer itself becomes unreliable.

E. Persistent Operation, Recovery, and Long-Term Growth

For a self-evolving autonomous robot, the ability to improve is meaningful only if the system can continue operating over long periods without losing its capacity for execution, recording, recovery, and further upgrading. Accordingly, the proposed model is designed not as a one-time adaptive program, but as a persistently running developmental system. Its objective is to remain alive in the loop of execution, reflection, revision, and redeployment, while continuously accumulating knowledge and verified methods over time.

The operational basis of this persistence is the heartbeat mechanism. Let the heartbeat at time t be denoted by $h_t \in \{0, 1\}$, where $h_t = 1$ indicates that the system is actively maintaining its execution cycle and $h_t = 0$ indicates loss of active progression. The heartbeat is not itself a complex task solver; instead, it triggers a recurring schedule for task execution, monitoring, learning, and recovery. This recurring process can be expressed as

$$\mathcal{CYCLE}_t = (\text{run}, \text{record}, \text{analyze}, \text{revise}, \text{validate})_t \,. \tag{29}$$

Thus, persistence means that the robot repeatedly re-enters this cycle rather than executing a task only once.

The scheduling process should itself be adjustable and learnable. Let Δ_t denote the scheduling interval used at time t. Then the heartbeat-driven loop may be written as

$$t_{k+1} = t_k + \Delta_k, \tag{30}$$

where Δ_k need not remain fixed. For example, a stable low-risk environment may allow a longer revision interval, while frequent failures or rapidly changing conditions may require more frequent checking and adaptation. Hence, persistence in the proposed model is not equivalent to rigid periodic execution; instead, it includes the ability to improve the execution schedule itself.

However, a persistent loop is useful only if failure can be detected and corrected. Therefore, the system includes explicit recovery monitoring. Let σ_t denote a health signal generated by the currently active upper-layer logic, such as a heartbeat flag update, a scheduler response, or a successful checkpoint write. Then a failure condition can be modeled as

$$\exists \tau > \tau_{max} \ : \ \sigma_{t:t+\tau} = 0, \tag{31}$$

meaning that no valid health signal has been observed within an allowable time window. Once this occurs, the system enters recovery mode.

Recovery may be performed by a simpler internal fallback mechanism or by an external watchdog-like mechanism. The key requirement is that failure detection must rely on a layer simpler than the one being recovered. Otherwise, if the same complex logic fails in both execution and detection, recovery

cannot be guaranteed. Let L_r denote the recovery layer and L_f denote the failed layer. Then the structural requirement is

$$\mathrm{Comp}(L_r) < \mathrm{Comp}(L_f), \tag{32}$$

where $\mathrm{Comp}(\cdot)$ denotes implementation complexity. This expresses the design principle that recovery logic should be simpler and therefore more reliable than the logic it supervises.

Once failure is detected, the system may restore a previously trusted implementation. Let $\mathcal{P}_t = \{C^{(0)}, C^{(1)}, \ldots, C^{(m)}\}$ denote the preserved set of recoverable historical versions available at time t. The recovery operator chooses a version

$$C_t^{rec} = \arg\max_{C \in \mathcal{P}_t} \Omega(C), \tag{33}$$

where $\Omega(C)$ denotes the estimated recoverability and trustworthiness of version C, based on past stability, retained essential functions, and availability of valid logs or connection capabilities. In many cases, the selected recovery version may be an older but more reliable implementation rather than the most recently generated one.

This preserved-version strategy is important not only for immediate fault handling, but also for developmental continuity. If a newly generated version fails, the system should not return to an empty state. Instead, it should fall back to a still-operational prior structure and continue learning from the failure. In this sense, failures are not terminal events but developmental evidence.

To formalize long-term growth, let $\mathcal{K}_t$ denote the accumulated developmental knowledge at time t, including verified code fragments, useful prompts, failed revisions, rollback cases, environmental regularities, and historical explanations. The growth process can then be written as

$$\mathcal{K}_{t+1} = \mathcal{K}_t \cup \Upsilon(\mathcal{G}_t, F_t, V_t), \tag{34}$$

where $\Upsilon(\cdot)$ denotes the extraction of reusable knowledge from complete generation information $\mathcal{G}_t$, environment feedback F_t, and validation results V_t. Unlike ordinary logs that merely record what happened, $\mathcal{K}_t$ represents information preserved for future improvement. It therefore functions as a developmental memory rather than passive storage.

This long-term memory allows the robot to gradually accumulate methods that were not predefined by designers. Over repeated interaction with the environment, the robot may preserve patterns of successful revision, identify classes of recurrent failure, discover more effective prompt structures, and develop increasingly robust operational logic. As a result, the system is intended to evolve not only in parameter values or temporary policies, but in its very manner of constructing future behavior.

IV. Verification

A. Verification Objectives

The verification has two objectives. First, we evaluate whether the proposed framework can maintain task success when the robot encounters progressively emerging constraints that are not covered by its initial predefined logic. Second, beyond task-level adaptation, we verify whether multiple layers of executable code can be replaced while preserving continued operation, and whether a faulty replacement can be detected and rolled back safely.

Accordingly, the verification is organized around three questions:

- Can the proposed robot outperform a fixed-code baseline when new task requirements appear?
- Can the system replace not only task-solving code but also higher-level operational logic?
- Can the system reject a faulty candidate and roll back to a previous working version without losing continued operation?

B. Experimental Setting

We constructed a sequential grid-world robotic benchmark consisting of 11 episodes. The robot acts in a discrete grid and outputs actions from

$$\mathcal{A} = \{\texttt{UP}, \texttt{DOWN}, \texttt{LEFT}, \texttt{RIGHT}\}. \tag{35}$$

The task requirements become progressively more complex across episodes. The task sequence includes:

- **Direct navigation**: reach the goal in an empty grid.
- **Obstacle avoidance**: the original downward-first route is blocked.
- **Key collection**: the robot must collect a key before entering the goal.
- **Forbidden red-cell avoidance**: the robot must avoid newly introduced forbidden cells.
- **Ordered waypoint visiting**: the robot must visit one or more waypoints before reaching the goal.
- **Energy-constrained navigation**: the robot must finish within a step budget.
- **Composite constraint handling**: the robot must simultaneously satisfy key, waypoint, obstacle, red-cell, and step-budget constraints.

The fixed-code baseline never changes its code during the experiment. The proposed robot starts from the same initial solver, but preserves complete generation information and uses LLM-based regeneration when update conditions are met. During the run, the evolving robot performed 7 update events, completed 11 layer replacement events in total, and experienced 1 rollback event. :contentReference[oaicite:0]index=0

1) Compared Methods: **Fixed-code baseline.** The baseline uses a manually predefined policy throughout the whole benchmark. It cannot revise either task logic or upper-level operational logic after deployment.

Proposed self-evolving robot. The proposed method preserves complete generation information, including task context, execution result, historical information, and current code, and uses this information to update executable logic. Importantly, the updated object is not restricted to the task solver alone. The framework also supports replacement of upper-layer logic such as update policy and scheduler behavior.

2) Evaluation Metrics: We use cumulative success rate as the primary performance metric. Let $s_i \in \{0, 1\}$ denote

TABLE I
CUMULATIVE ACCURACY SEQUENCE OVER THE 11 EPISODES

Episode	Fixed-Code Baseline	Evolving Robot
1	1.0000	1.0000
2	0.5000	1.0000
3	0.3333	1.0000
4	0.2500	1.0000
5	0.4000	1.0000
6	0.3333	1.0000
7	0.2857	1.0000
8	0.2500	1.0000
9	0.2222	1.0000
10	0.3000	1.0000
11	0.2727	1.0000

whether episode i is completed successfully. Then the cumulative accuracy up to episode t is

$$\mathrm{Acc}(t) = \frac{1}{t}\sum_{i=1}^{t} s_i. \tag{36}$$

To evaluate replaceability and safety, we additionally record:

- the number of update events,
- the total number of accepted layer replacements,
- and the number of rollback events after faulty replacement attempts.

C. Capability Adaptation Results

Figure 1 plots the cumulative accuracy curves of the fixed-code baseline and the proposed evolving robot. The figure shows a clear divergence between the two methods as task requirements become more complex. The fixed-code baseline drops rapidly after the first few episodes, whereas the proposed method maintains perfect performance throughout the whole sequence.

The fixed-code baseline succeeds only in Episodes 0, 4, and 9, and fails in the other 8 episodes when new constraints are introduced. Its final cumulative accuracy is

$$\mathrm{Acc}_{fixed}(11) = 0.2727. \tag{37}$$

By contrast, the proposed self-evolving robot succeeds in all 11 episodes, yielding

$$\mathrm{Acc}_{prop}(11) = 1.0. \tag{38}$$

Thus, the absolute improvement over the fixed-code baseline is

$$\Delta = 1.0 - 0.2727 = 0.7273. \tag{39}$$

The cumulative accuracy results are listed in Table I. As the task sequence becomes progressively more complex, the fixed-code baseline shows a clear downward trend, whereas the evolving robot maintains perfect cumulative accuracy throughout the whole run.

As shown in Figure 1, the fixed-code baseline does not fail gradually in a smooth optimization sense; instead, it repeatedly breaks when a new constraint invalidates its predefined path logic. By contrast, the proposed method remains stable even when obstacle handling, key collection, forbidden-cell avoidance, waypoint ordering, and composite constraints are introduced. This visual contrast directly supports the claim that preserving complete generation information allows the robot to extend its logic beyond the scope of the initial hand-written code.

The failure modes of the fixed-code baseline are also informative. On obstacle tasks, it hits obstacles immediately at step 0. On key tasks, it reaches the goal without collecting the required key. On red-cell tasks, it steps onto forbidden cells. On waypoint tasks, it reaches the goal before visiting all required waypoints. On the final composite task, it again fails under multi-constraint coupling. These observations show that a fixed predefined policy quickly becomes inadequate once the environment introduces new requirements.

D. Verification of Multi-Layer Code Replaceability

Figure 2 summarizes the accepted replacements across different layers together with the rollback event. The figure shows that replaceability is not confined to the task solver. Instead, successful runtime replacement also occurred in higher-level control logic.

The log shows that the evolving robot successfully replaced multiple types of executable logic:

- **Task solver replacement.** Task-solver replacements were accepted at Episodes 0, 1, 3, 5, 7, 8, and 9.
- **Update-policy replacement.** Update-policy replacements were accepted at Episodes 1 and 7.
- **Scheduler replacement.** A scheduler replacement was accepted at Episode 9.

Thus, the experiment does not only verify that the robot can regenerate task-specific behavior. It also verifies that higher-level operational logic, including the policy that decides whether to update and the scheduler that governs update allowance, can itself be replaced during runtime while continued operation is preserved.

In total, the system completed 11 layer replacement events across the run. This number is larger than the number of update events (7), which indicates that a single update event may modify more than one layer. For example, at Episode 1, both the `update_policy` and the `task_solver` were successfully replaced, and at Episode 9, both the `scheduler` and the `task_solver` were successfully replaced.

Figure 2 provides an intuitive structural view of this result. The dominant bar corresponds to task-solver replacement, which is expected because task-solving logic is the most directly challenged by new environmental constraints. However, the presence of accepted update-policy and scheduler replacements is crucial, because it demonstrates that the system can revise not only "what to do" for a task, but also "when to update" and "how to organize the updating process." This observation directly supports the multi-layer replaceability claim of the proposed architecture.

E. Verification of Failed Replacement and Rollback

The experiment also verifies the safety mechanism for faulty code replacement. At Episode 5, after a valid task-solver replacement was accepted, a `task_solver_faulty_attempt` was recorded with `accepted=False` and `rolled_back=True`.

TABLE II
EPISODE-LEVEL RESULTS OF THE FIXED-CODE BASELINE AND THE PROPOSED EVOLVING ROBOT

Episode	Task Type	Fixed-Code Baseline	Proposed Method	Notes
0	direct	Success	Success	initial update accepted
1	obstacle	Fail	Success	update policy and task solver replaced
2	obstacle	Fail	Success	no new update needed
3	key	Fail	Success	task solver replaced
4	key	Success	Success	no new update needed
5	red	Fail	Success	task solver replaced; faulty attempt rolled back
6	red	Fail	Success	no new update needed
7	waypoint	Fail	Success	update policy and task solver replaced
8	waypoint	Fail	Success	task solver replaced
9	energy	Success	Success	scheduler and task solver replaced
10	composite energy task	Fail	Success	no additional update required

This event provides direct evidence of failed replacement handling:

1) a candidate replacement was attempted,
2) the candidate was identified as unsuitable,
3) the replacement was not kept as the active version,
4) and the system rolled back successfully.

The global summary reports

$$\texttt{rollback_count} = 1, \tag{40}$$

which confirms that rollback was not merely implemented in principle but was actually triggered during the experimental run.

Equally important, the rollback did not break future operation. After the faulty attempt at Episode 5, the robot still succeeded in Episodes 6, 7, 8, 9, and 10. Therefore, the rollback mechanism preserved both immediate task continuity and future upgradability.

Figure 2 also includes the rollback count. Although only one rollback event occurred in this experiment, its significance is structural rather than statistical. The event shows that the framework does not rely on blindly trusting every generated candidate. Instead, it maintains a validation-and-recovery path, which is essential for any self-evolving system that aims to preserve long-term survivability during runtime code replacement.

F. Discussion

Taken together, Figure 1, Figure 2, and Table II support three main conclusions.

First, the proposed method substantially outperforms a fixed-code robot when new requirements are introduced. The fixed-code baseline ends with a cumulative accuracy of only 0.2727, whereas the evolving robot maintains 1.0 across all episodes.

Second, the experiment verifies *multi-layer replaceability*. The runtime log shows successful replacement not only of task-solving code, but also of update-policy logic and scheduler logic. This is important because the paper claims that autonomous growth should extend beyond a single business-logic module.

Third, the experiment verifies *safe failed replacement and rollback*. A faulty replacement attempt occurred and was rolled back, and the system continued to operate successfully afterward. Hence, the experiment supports the core claim

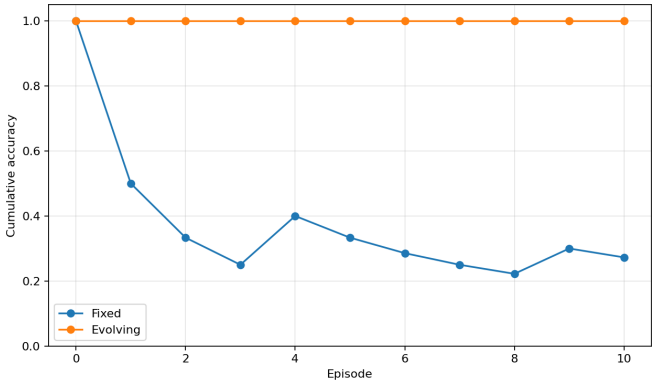


Fig. 1. Cumulative accuracy curves of the fixed-code baseline and the proposed evolving robot across the 11 episodes. The fixed-code baseline degrades as new constraints are introduced, while the proposed method maintains perfect cumulative accuracy through iterative code replacement.

that self-evolving code must be accompanied by survivability-preserving replacement mechanisms.

Overall, the current verification demonstrates not only task adaptation but also runtime replaceability of multiple code layers together with rollback-based safety. The figures make these two properties visually explicit: Figure 1 shows the capability advantage over the fixed baseline, while Figure 2 shows that this advantage is achieved through actual code replacement across multiple layers rather than through a single static predesigned solver. These results are therefore consistent with the main thesis of the paper: an autonomous robot can preserve complete generation information, replace its own executable logic across multiple levels, and continue growing without depending on permanently fixed predefined task code.

V. CONCLUSION

This paper studied how autonomous robots can move beyond reliance on fixed predefined logic and continue improving their executable behavior through code-level self-evolution. To address this problem, we proposed a complete-generation-information-driven self-learning framework in which the robot preserves not only its current code, but also the full generation and execution chain behind that code, including prompts, goals, runtime traces, feedback, and historical logs. Based on this preserved information, the robot can revise its own logic through LLM interaction, support

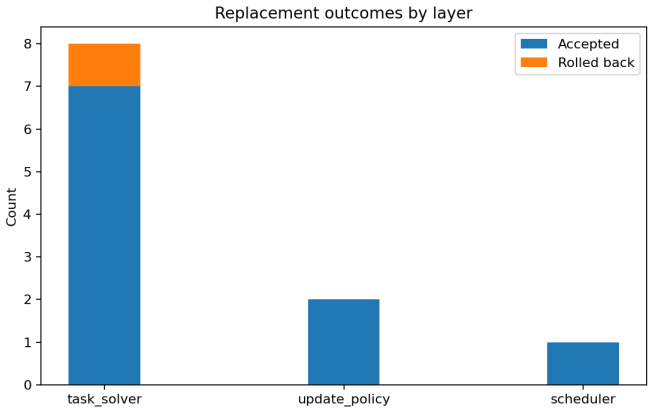


Fig. 2. Counts of accepted replacements across different code layers together with the rollback event. The figure shows that runtime replacement occurred not only in the task solver but also in update-policy and scheduler logic, and that a faulty replacement attempt was safely rolled back.

runtime replacement across multiple code layers, and maintain survivability through safe rollback. Experimental results on progressively changing robotic tasks showed that the proposed method substantially outperformed a fixed-code baseline. In the reported benchmark, the proposed system achieved a final cumulative success rate of 1.0, compared with 0.2727 for the fixed-code robot, while performing 7 update events, 11 successful layer replacement events, and 1 rollback event.

In future work, we will extend the framework to more complex embodied environments, longer developmental horizons, and more diverse forms of runtime uncertainty, including dynamic environments, partially observable settings, and richer interactions between local and remote intelligence.

REFERENCES

[1] X.-K. Wu, M. Chen, W. Li, R. Wang, L. Lu, J. Liu, K. Hwang, Y. Hao, Y. Pan, Q. Meng *et al.*, "Llm fine-tuning: Concepts, opportunities, and challenges," *Big Data and Cognitive Computing*, vol. 9, no. 4, p. 87, 2025.

[2] J. Cordeiro, S. Noei, and Y. Zou, "Llm-driven code refactoring: Opportunities and limitations," in *2025 IEEE/ACM Second IDE Workshop (IDE)*. IEEE, 2025, pp. 32–36.

[3] W. Chen, W. Chi, S. Ji, H. Ye, J. Liu, Y. Jia, J. Yu, and J. Cheng, "A survey of autonomous robots and multi-robot navigation: Perception, planning and collaboration," *Biomimetic Intelligence and Robotics*, vol. 5, no. 2, p. 100203, 2025.

[4] I. Singh, V. Blukis, A. Mousavian, A. Goyal, D. Xu, J. Tremblay, D. Fox, J. Thomason, and A. Garg, "Progprompt: Generating situated robot task plans using large language models," *arXiv preprint arXiv:2209.11302*, 2022.

[5] Y. J. Ma, W. Liang, G. Wang, D.-A. Huang, O. Bastani, D. Jayaraman, Y. Zhu, L. Fan, and A. Anandkumar, "Eureka: Human-level reward design via coding large language models," *arXiv preprint arXiv:2310.12931*, 2023.

[6] F. Zeng, W. Gan, Y. Wang, N. Liu, and P. S. Yu, "Large language models for robotics: A survey," *arXiv preprint arXiv:2311.07226*, 2023.

[7] V. Kumar, "Act-observe-rewrite: Multimodal coding agents as in-context policy learners for robot manipulation," *arXiv preprint arXiv:2603.04466*, 2026.

[8] C. Y. Kim, N. T. White, E. He, F. Sala, and B. Mutlu, "Robocritics: Enabling reliable end-to-end llm robot programming through expert-informed critics," in *Proceedings of the 21st ACM/IEEE International Conference on Human-Robot Interaction*, 2026, pp. 914–923.

[9] X. Yin, X. Wang, L. Pan, L. Lin, X. Wan, and W. Y. Wang, "Gödel agent: A self-referential agent framework for recursive self-improvement," 2025. [Online]. Available: https://arxiv.org/abs/2410.04444

[10] H.-a. Gao, J. Geng, W. Hua, M. Hu, X. Juan, H. Liu, S. Liu, J. Qiu, X. Qi, Y. Wu *et al.*, "A survey of self-evolving agents: What, when, how, and where to evolve on the path to artificial super intelligence," *arXiv preprint arXiv:2507.21046*, 2025.

[11] J. Zhang, S. Hu, C. Lu, R. Lange, and J. Clune, "Darwin godel machine: Open-ended evolution of self-improving agents," *arXiv preprint arXiv:2505.22954*, 2025.

[12] M. Robeyns, M. Szummer, and L. Aitchison, "A self-improving coding agent," *arXiv preprint arXiv:2504.15228*, 2025.

PLACE PHOTO HERE

Hong Su received the MS and PhD degrees, in 2006 and 2022, respectively, from Sichuan University, Chengdu, China. He is currently a researcher of Chengdu University of Information Technology Chengdu, China. His research interests include blockchain, large language model and human simulation computing.