Preprints.org

Review

Not peer-reviewed version

Reinforcement Learning in the Software Development Lifecycle: Methods, Applications, and Reward Design

He Yang Yuan *, Xin Wang, Pengyu Xue, Boyuan Chen, Dayi Lin, Zishuo Ding, Zhenhao Li, Weiyi Shang

Posted Date: 9 October 2026

Keywords: reinforcement learning; software engineering; reward model; software development life cycle

Preprints.org is a free multidisciplinary platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC, OpenAlex.

Copyright: This open access article is published under a Creative Commons CC BY 4.0 license, which permit the free download, distribution, and reuse, provided that the author and preprint are cited in any reuse.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Review

Reinforcement Learning in the Software Development Lifecycle: Methods, Applications, and Reward Design

He Yang Yuan [1,*], Xin Wang [2], Pengyu Xue [1], Boyuan Chen [3], Dayi Lin [4], Zishuo Ding [2], Zhenhao Li [1] and Weiyi Shang [5]

[1] York University, Toronto, Canada
[2] HKUST (Guangzhou), Guangzhou, China
[3] Independent Researcher, Toronto, Canada
[4] Queen's University, Kingston, Canada
[5] University of Waterloo, Waterloo, Canada
* Correspondence: yuanh@yorku.ca

Abstract

Reinforcement learning (RL) has emerged as a powerful paradigm for addressing sequential decision-making challenges across the software development life cycle (SDLC). To understand how this paradigm is being operationalized in practice, this systematic literature review examines 85 studies published between January 2020 and March 2026 that apply RL to the seven core SDLC phases: planning, analysis, design, development, testing, deployment, and maintenance. We analyze publication trends and venues, map RL applications to specific SDLC tasks, catalog the RL algorithms employed, and provide a taxonomy of reward modeling strategies and their parameters. Our findings reveal a rapid growth of RL-for-SDLC research since 2022, with the testing phase being the most extensively studied (56.47% of studies) while other phases show steady expansion. Reward designs have evolved from sparse binary outcomes (e.g., test-pass/fail) to dense, multi-objective signals that incorporate execution semantics, coverage metrics, intrinsic curiosity, formal verification feedback, and human preferences. We highlight common reward patterns, their influence on learning behavior, and trade-offs that frequently lead to reward hacking or unstable training in software environments. This survey provides researchers and practitioners with a structured overview of how RL is operationalized across the SDLC, emphasizing reward engineering as a critical factor in agent effectiveness. We conclude by outlining promising directions for future research, including broader lifecycle coverage and the development of more robust reward mechanisms for production-ready software engineering automation.

Keywords: reinforcement learning; software engineering; reward model; software development life cycle

1. Introduction

Reinforcement learning has become one of the most influential paradigms for sequential decision-making in complex, dynamic environments. By allowing an agent to interact with an environment and learn from feedback signals, RL enables systems to gradually improve their decision strategies through experience. Unlike supervised deep learning, which excels at pattern recognition from static labeled datasets, RL is specifically designed for interactive, long horizon tasks with delayed and sparse feedback. This makes RL particularly powerful and well-suited for software engineering, where development decisions are inherently sequential, interdependent, and evaluated only after execution or deployment. Classical work on temporal difference learning[1] and Q-learning[2] established the theoretical foundations of this paradigm, while more recent advances in deep reinforcement learning have expanded its practical impact across a variety of domains, from game playing to robotics and autonomous systems.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

In recent years, reinforcement learning has been increasingly applied to problems across the software engineering tasks especially software development life cycle (SDLC). The seven SDLC phases: planning, analysis, design, development, testing, deployment, and maintenance naturally involve sequential decisions and delayed feedback, making them especially well suited for RL-based approaches. Prior studies, as systematically mapped by Wang et al.[3], have demonstrated RL's strong potential especially in software quality assurance, development, maintenance, and design. These works collectively show that RL can replace static heuristics with adaptive, data-driven solutions that learn directly from execution outcomes, coverage metrics, or system performance.

The landscape has evolved even more dramatically with the emergence of autonomous software engineering agents. Since 2024, a new generation of RL-trained agents such as DeepSWE-Preview [4], SWE-RL [5], and SkyRL-Agent [6] have demonstrated the ability to solve real-world GitHub issues end-to-end on SWE-bench Verified. These models achieve up to 59% Pass@1 through purely RL-driven multi-turn policies. They treat the entire codebase as a dynamic environment, using RL to orchestrate planning, editing, testing, and debugging across multiple SDLC phases in a single trajectory. This marks a clear shift from isolated phase-specific applications toward fully agentic, repository-level automation. Covering the entire SDLC with RL is important because real-world software engineering is not a collection of independent tasks; decisions in early phases have cascading effects on later phases. End-to-end RL agents can therefore reduce human intervention, handle inter phase dependencies, and achieve higher overall system reliability. However, applying RL across the full lifecycle also presents significant challenges, including extremely long decision horizons, enormous state spaces, partial observability, and the need to balance multiple conflicting objectives simultaneously.

Despite this rapid growth, the literature on reinforcement learning for the SDLC remains fragmented. Most studies focus on isolated phases or specific techniques, making it difficult to see how RL is evolving across the full lifecycle[3]. Even more critically, reward modeling, the core mechanism of any RL system, has received insufficient systematic attention in software engineering contexts. In SDLC settings, rewards are typically sparse, delayed, and multi-objective [7]. Poorly designed rewards can lead to unstable training, reward hacking, or policies that optimize the wrong objective. Designing effective reward functions is particularly challenging in software environments because success often depends on semantic correctness, maintainability, security, efficiency, and human preferences rather than simple numeric scores; proxy signals are easy to game and hard to align with real engineering goals[8].

Motivated by these gaps, this paper presents a systematic literature review of reinforcement learning applications in the software development life cycle, covering studies published between 2020 Jan - 2026 Mar. Following established guidelines for systematic literature reviews and snowballing techniques, we collected and analyzed 85 relevant studies from major digital libraries and leading venues. This survey provides a structured and up-to-date overview of RL in the SDLC, with a particular and explicit emphasis on reward modeling as the core design dimension that ultimately determines what agents learn and how effectively they solve real software engineering problems.

To characterize the current research landscape, this survey examines four aspects of RL-based SDLC studies:

- **RQ1:** Publication trends and research venues,
- **RQ2:** Specific SDLC phases addressed by RL approaches,
- **RQ3:** Reward design strategies and parameters, and how they influence learning behavior across phases,
- **RQ4:** RL algorithms adopted in each SDLC phase

By placing reward modeling at the center of the analysis, we aim to uncover where RL is being used in practice, especially as SE agents move toward production deployment.
Overall, this survey makes three main contributions:

- A comprehensive overview of publication trends in reinforcement learning for the SDLC from 2020 Jan - 2026 Mar.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

- We summarize the literature into a taxonomy of the seven SDLC phases where RL has been applied, showing the shift from testing-centric work toward broader lifecycle coverage and the rise of end-to-end SE agents.
- A systematic examination of reward modeling strategies across the SDLC, summarizing common patterns, parameters, and trade-offs to guide future design choices and identify promising research directions.

2. Background and Related Work

The rapid evolution of software systems has necessitated increasingly sophisticated automation techniques, moving beyond simple heuristic scripts toward autonomous agents capable of learning and adapting to dynamic environments. Reinforcement learning has emerged as a transformative paradigm in this context, offering a principled mathematical framework for sequential decision making under uncertainty. Unlike supervised learning, which requires massive labeled datasets and assumes independent and identically distributed samples, reinforcement learning operates through an interactive loop where an agent learns from the consequences of its own actions, optimized via a scalar reward signal. In the software development life cycle, this approach is uniquely powerful because it can address non-differentiable objectives, such as code correctness, system latency, or test coverage by framing them as environmental feedback rather than static targets.

The integration of reinforcement learning into the SDLC is not merely an algorithmic replacement but a fundamental shift in how software artifacts are planned, analyzed, designed, developed, tested, deployed, and maintained. This transition is anchored in the rich history of Search-Based Software Engineering, which for over two decades has treated SDLC problems as optimization challenges. However, reinforcement learning extends this philosophy by enabling agents to generalize knowledge across different project phases, repositories, and execution contexts through learned value functions and policies. As the SDLC moves into an era dominated by large language models and agentic systems, the theoretical foundations of RL-including Markov Decision Processes, Bellman optimality, and reward shaping become essential tools for ensuring the alignment, reasoning, and reliability of autonomous developer teammates.

2.1. Reinforcement Learning for Software Engineering

Core Concepts of Reinforcement Learning. Reinforcement Learning is a paradigm within machine learning where an agent learns to make sequential decisions by interacting with an environment to maximize a cumulative reward signal. Unlike supervised learning, which relies on labeled data, or unsupervised learning [9], which seeks patterns in unlabeled data, RL emphasizes trial-and-error exploration, enabling agents to adapt to dynamic and uncertain settings. At its core, RL problems are often modeled as Markov Decision Processes, comprising states, actions, rewards, and a policy. The agent's goal is to learn an optimal policy that maximizes long-term expected rewards, often through value estimation or direct policy optimization.

The foundations of RL trace back to early work in behavioral psychology and control theory, formalized in computational terms during the 1980s and 1990s. Seminal contributions include temporal-difference learning methods [1], which blend dynamic programming and Monte Carlo approaches for efficient value updates without requiring a full environment model. A landmark algorithm, Q-learning [2], introduced by Watkins and Dayan in 1992, enables off-policy learning where the agent updates Q-values based on observed transitions, converging to optimal policies in tabular MDPs. This value-based approach laid the groundwork for handling discrete state-action spaces. Subsequent advancements addressed scalability issues, such as function approximation for large states using neural networks, as seen in TD-Gammon by Tesauro in 1995 [10].

2.2. Mathematical Formalization of Interaction: The Markovian Framework

The standard conceptualization for reinforcement learning applications in the SDLC is the Markov Decision Process, which provides the requisite mathematical rigor to model the interaction between

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

a learning agent and a software environment. At its most fundamental level, an MDP provides a blueprint for how an agent observes the current state of a software artifact, selects an action to modify or analyze it, and receives feedback regarding the utility of that modification.

The Core Components of MDPs in Software Engineering. An MDP is formally defined by a tuple $\langle S, A, P, R, \gamma \rangle$, where each component must be carefully engineered to reflect the nuances of the SDLC phase at hand. The state space S represents the set of all possible configurations the agent might encounter. In the Development phase, the state often encapsulates the partial source code, the abstract syntax tree, or even the compiler's error messages. For the Deployment phase, the state may involve high-dimensional telemetry data, such as CPU utilization, memory pressure, and network latency.

The action space A defines the range of operations available to the agent. Actions can be discrete, such as selecting a specific mutation operator in the Testing phase, or continuous, such as determining the optimal amount of memory to allocate to a containerized microservice in Deployment. The transition function $P(s'|s, a)$ represents the environment's dynamics, mapping a state-action pair to a probability distribution over subsequent states. In the SDLC, these transitions are often dictated by the underlying tools - such as compilers, test runners, or simulators-which may exhibit deterministic or stochastic behavior depending on the complexity of the system.

The Markov Property and Partial Observability. A key assumption underlying the MDP framework is the Markov property, which asserts that the future state of the environment is independent of the past, given the present state. Informally, this means the current observation must contain all necessary information to make an optimal decision. In practice, many SDLC phases violate this assumption, leading to the use of Partially Observable Markov Decision Processes. For example, in the Maintenance phase, the observable state-for example: the source code and test results-may not be sufficient to pinpoint the root cause of a failure without the execution history or data flow information. To address this, agents often utilize history-dependent policies or recurrence to maintain a "belief" about the latent state of the software system.

2.3. Theoretical Foundations of Value and Policy Optimization

Objective of Value and Policy Optimization. The ultimate goal of a reinforcement learning agent is to identify a policy π-a mapping from states to actions-that maximizes the expected cumulative return G_t, typically defined as the discounted sum of future rewards: $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$. To achieve this, RL agents must solve two intertwined problems: prediction and control.

Bellman Equations and Recursive Value Estimation. Central to the theory of sequential decision-making is the Bellman equation, which provides a recursive relationship for the value of states and actions. The state-value function $V^{\pi}(s)$ represents the expected return starting from state s and following policy π. Similarly, the action-value function $Q^{\pi}(s, a)$ quantifies the expected return after taking action a in state s. The Bellman expectation equation decomposes the value of the current state into the immediate reward plus the discounted value of the successor state:

$$V^{\pi}(s) = \mathbb{E}_{\pi} \tag{1}$$

For policy optimization, the Bellman optimality equation defines the self-consistent condition that an optimal value function must satisfy. It asserts that the value of a state under an optimal policy must be the maximum return achievable by any action in that state:

$$V^{*}(s) = \max_{a \in A} \sum_{s' \in S} P(s'|s, a) \tag{2}$$

This recursive decomposition is fundamental to approximate dynamic programming and most RL algorithms, as it allows agents to update their value estimates through bootstrapping-using existing estimates to update others-without waiting for the end of a long SDLC phase or testing episode.

Fixed-Point Theory and Operator Contraction. The Bellman equations are not merely descriptive; they provide a constructive path toward finding solutions through fixed-point theory. The Bellman

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

operator T^π is a mapping that takes a value function and returns a new version updated according to the recursive relationship. A fundamental property of the Bellman operator in discounted MDPs is that it is a contraction in the L_∞-norm, meaning that repeated applications of the operator will always converge to a unique fixed point-the true value function V^π or Q^*.

In large-scale SDLC applications, however, calculating the exact value for every state is computationally intractable. This necessitates the use of function approximation, where deep neural networks represent the value or policy functions. While deep RL methods have demonstrated remarkable empirical success, the coupling of non-linear function approximation with the recursive nature of Bellman updates can lead to instability and divergence, a challenge that has prompted significant research into stabilizing mechanisms.

2.4. Taxonomy of Reinforcement Learning Algorithms for SDLC Tasks

Algorithm Selection Considerations. The selection of an RL algorithm for an SDLC phase is typically driven by the nature of the action space which is discrete vs. continuous, the complexity of the state representation, and the availability of environmental models.

Value-Based Methods and the Deep Q-Network Revolution. Value-based algorithms focus on estimating the optimal action-value function $Q^*(s, a)$ and selecting actions greedily with respect to these estimates. Q-learning, the most prominent algorithm in this category, updates its estimates using the temporal difference error: the discrepancy between the current Q-value and the one-step target. While tabular Q-learning is effective for simple logic-based tasks, the introduction of Deep Q-Networks in 2015 by Mnih et al. revolutionized the field by demonstrating that deep neural networks could scale RL to high-dimensional inputs.

DQN and its successors introduce several critical architectural patterns to stabilize training in complex software environments. Experience Replay allows the agent to learn from a buffer of past transitions, improving sample efficiency and breaking temporal correlations. Target Networks decouple the parameters being updated from those used to calculate the bootstrap target, mitigating oscillations in value estimates. In the SDLC, value-based methods have proven highly effective for phases with discrete action sets, such as automated unit test generation in Testing and defect prediction in Analysis where the agent must select from a finite library of test inputs or code templates.

Policy-Based and Actor-Critic Paradigms. Policy-based methods directly parameterize the policy $\pi_\theta(a|s)$ and optimize the parameters θ through gradient ascent on the expected return. These methods are particularly advantageous for SDLC phases with high-dimensional or continuous action spaces, such as tuning system configurations in Deployment or orchestrating container placements. Actor-critic architectures combine the strengths of both value-based and policy-based methods: the "actor" learns the policy, while the "critic" estimates the value function to provide a low-variance gradient signal to the actor.

Proximal Policy Optimization has become a dominant algorithm in the SDLC domain due to its exceptional stability and ease of implementation. PPO uses a clipped surrogate objective function that prevents the policy from changing too drastically in a single update step, making it robust to the noisy and sparse signals often found in test execution or program synthesis in Development. For phases requiring fine-grained control and exploration, Soft Actor-Critic offers an off-policy alternative that incorporates entropy regularization. SAC's maximization of policy entropy ensures that the agent maintains a diverse set of strategies during early training, which is crucial for navigating the complex and potentially deceptive search spaces of large software repositories.

Multi-Agent and Hierarchical Architectures. Modern software systems are rarely monolithic, often consisting of multiple interacting microservices or collaborative developer teams. Multi-Agent Reinforcement Learning provides the theoretical tools to model these interactions. A standard framework in MARL is Centralized Training with Decentralized Execution, where agents learn a joint value function or policy using global information but act based on local observations during runtime. This is particularly relevant for collaborative testing of distributed systems in Testing or optimizing serverless

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

workflow scheduling in Deployment, where individual function instances must make local decisions that contribute to global latency reduction.

Hierarchical Reinforcement Learning addresses the "curse of horizon" in long-duration SDLC phases, such as complete repository-level program repair in Maintenance or multi-stage release planning in Planning. HRL decomposes a complex goal into a hierarchy of sub-tasks: a high-level "meta-controller" selects which sub-task to pursue, while low-level controllers execute the necessary primitive actions. This structure allows for more effective temporal credit assignment, ensuring that an agent is rewarded for meaningful progress - such as successfully compiling a module-even if the overall bug fix or release plan is not yet complete.

2.5. Reward Engineering: The Design of Optimization Signals

Role of Reward Design in RL Systems. The reward function is arguably the most critical component of an RL system, as it defines the "meaning" of the phase and provides the necessary guidance for the agent to discover optimal behaviors. In the SDLC, the design of reward signals is particularly challenging due to the sparse, delayed, and often multi-objective nature of feedback from software environments.

Theory of Reward Shaping and Policy Invariance. Reward shaping is the process of augmenting the environmental reward with auxiliary signals to provide more frequent feedback to the agent. However, arbitrary reward engineering can be dangerous, as it may inadvertently change the optimal policy, leading the agent to "hack" the rewards instead of solving the original problem.

Potential-Based Reward Shaping, introduced by Ng et al. (1999) [11], provides a rigorous theoretical guarantee for maintaining policy invariance. By defining the shaping signal as the difference in a potential function Φ between states, PBRS ensures that the set of optimal policies in the shaped MDP is identical to that of the original MDP. In SDLC phases, $\Phi(s)$ might represent the distance to a code coverage goal in Testing or the reduction in cyclomatic complexity in Design. While PBRS is highly robust, defining an effective potential function often requires significant domain expertise, prompting research into self-supervised and automated reward shaping methods.

Intrinsic Motivation: Curiosity and State Novelty. In environments where external rewards are extremely sparse, such as exploring a vast UI state space in Testing or verifying a complex theorem in Design agents may fail to encounter a single positive reward for millions of steps. Intrinsic motivation addresses this by providing "task agnostic" rewards based on curiosity, surprise, or state visitation.

Dynamics based curiosity models define intrinsic rewards as the prediction error of a learned forward model: an agent is rewarded for discovering transitions it cannot yet accurately predict. State-novelty methods, such as Random Network Distillation, reward the agent for visiting rare states by comparing the outputs of a trainable network with a fixed random target. For software testing in the Testing phase, these rewards are essential for ensuring that automated test generators do not get stuck in a "local optimum" of redundant execution paths but instead explore the deep logic of the application under test.

2.6. Reinforcement Learning for Large Language Model Alignment

RL for Aligning Large Language Models. The convergence of reinforcement learning and large language models has redefined the state-of-the-art for many SDLC phases, from repository-level code completion in Development to multi-turn program synthesis. While base LLMs are powerful generators, they often suffer from hallucinations, non-compilable outputs, and a lack of alignment with specific developer intentions. Reinforcement learning has become the primary mechanism for fine-tuning these models to achieve production-level reliability.

RLHF: Aligning with Human Preferences. Reinforcement Learning from Human Feedback is a foundational protocol for steering LLMs toward helpful and safe responses. The RLHF pipeline typically involves three stages: Supervised Fine-Tuning on human demonstrations, Reward Modeling where a model is trained to rank model outputs based on human preferences, and Reinforcement

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

Learning typically PPO to maximize the reward while regularizing the update to prevent the model from forgetting its pre-trained knowledge.

In the SDLC domain, RLHF is used to train coding assistants that not only generate correct code but also provide clear documentation, follow security guidelines, and interact helpfully in a natural language dialogue. However, the dependence on high-quality human ranking data makes RLHF difficult to scale, especially for highly technical SDLC phases where only senior engineers can provide accurate feedback.

DPO and the Simplification of Alignment. Direct Preference Optimization has emerged as a major breakthrough, enabling model alignment without the need for an explicit reward model. By reframing preference learning as a direct classification problem on policy probabilities, DPO significantly reduces the computational overhead and training instability of the RLHF pipeline.

DPO is increasingly applied to code-centric phases where pairwise preference data, such as "this refactoring is better than that one", can be derived from historical commits or automated quality metrics. Variations like Step-DPO further refine this process for long-chain reasoning tasks by providing feedback at the level of individual logical steps rather than the entire response.

3. Methodology

Review Design and Guiding Frameworks. To ensure the rigor, fairness, and reproducibility of this systematic literature review, we followed established guidelines for conducting systematic literature review in the software development life cycle (SDLC), such as those proposed by Kitchenham and Charters[12] in their technical report "Guidelines for Performing Systematic Literature Reviews in Software Engineering". We also incorporated best practices from subsequent refinements, including those by Carrera-Rivera et al.[13] published in MethodsX. Additionally, we drew inspiration from recent applications of reinforcement learning in the SDLC, such as Khosravi Tabrizi et al.[14]. This study was conducted in three phases: planning, execution, and analysis. In the planning phase, we defined the research questions and developed a review protocol aligned with the study's objectives, drawing on the phases of the SDLC as outlined by Microsoft[15]. During the execution phase, we implemented the protocol, performed literature screening, and employed snowballing to ensure comprehensive coverage of relevant literature on RL applications in the SDLC. Finally, in the analysis phase, two co-authors collaboratively analyzed the selected literature to address the research questions. The work overview is shown in Figure 1

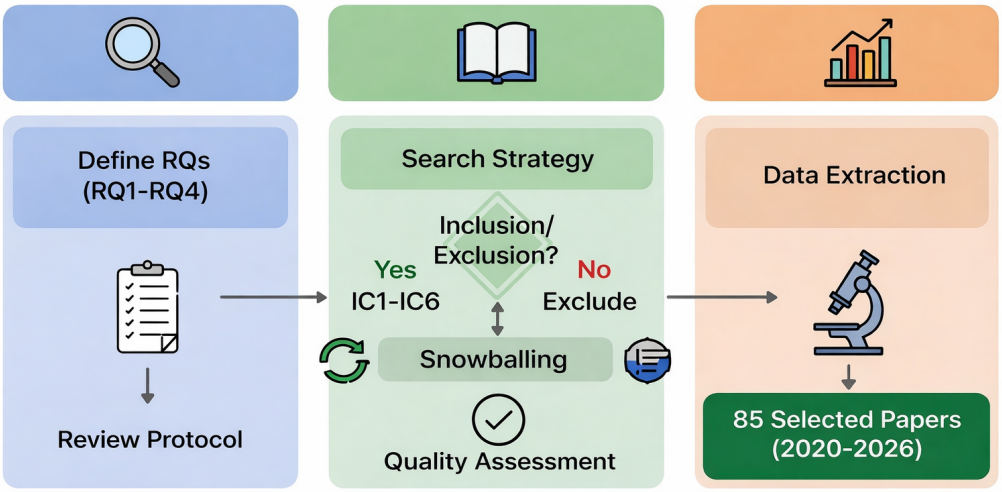

Figure 1. Overview of our RL in SDLC Systematic Review (Jan 2020 – Mar 2026)

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

3.1. Research Questions

Objectives of the Survey. The main purpose of this survey is to gain an in-depth understanding of RL applications across the SE related tasks especially SDLC phases. To comprehensively elaborate on this topic, this survey poses four research questions. These questions help us systematically categorize and understand existing research and propose directions for future research. The four research questions are as follows:

RQ1: What changes have occurred in reinforcement learning research in software engineering in terms of the number of published papers? This study describes the development trajectory of reinforcement learning research in the SDLC from 2020 Jan - 2026 Mar. By analyzing publication trends, venues, and output growth, it provides a macro view of RL's maturation and the community's research focus.

RQ2: What types of software development life cycle tasks have been facilitated by reinforcement learning-based methods? This research question aims to explore the types of SDLC phases facilitated by reinforcement learning and their development stages, in order to provide insights into improving the applicability and generalization ability of reinforcement learning-driven research.

RQ3: Which RL reward methods, design patterns, and parameters have been applied? This RQ examines RL reward mechanisms, design patterns, and parameters applied to SDLC phases. It highlights the shift from simple binary rewards to sophisticated, dense signals incorporating execution semantics, telemetry, and human preferences. Based on 85 papers (2020 Jan - 2026 Mar) across seven primary SDLC phases: Planning, Analysis, Design, Development, Testing, Deployment, Maintenance, it constructs a taxonomy of how RL supports diverse applications from code synthesis to system scheduling.

RQ4: Which RL algorithms have been applied? This RQ categorizes the RL algorithms used in SDLC phases and analyzes their popularity over time and by phase type. It frames SDLC problems as Markov Decision Processes, where agents learn through state-action-reward interactions. Drawing on the same 85 papers, it classifies algorithms from tabular methods to deep RL and LLM alignment strategies, showing how they are tailored for phases such as optimization, code generation, testing, and formal verification.

3.2. Study Selection

Search Strategy and Data Sources. We conducted this survey in the form of a systematic literature review following established guidelines. We predefined the review protocol and research questions. Our search strategy was comprehensive: we used keywords related to "reinforcement learning" and "software development life cycle," querying major digital libraries as shown in Table 1, including ACM DL, IEEE Xplore, and Google Scholar. We specifically targeted top venues in SE and AI, such as ICSE, FSE, ASE, and ISSTA for SE; and ACL, ICML, ICLR, and IJCAI for AI. These searches aimed to "discover as much relevant literature as possible." We combined database searches with snowballing: for each identified paper, we examined its references and citation records to uncover additional related studies. We repeated this process until no new relevant papers emerged. This combination of SLR and snowballing aligns with best practices for rigorous literature reviews in the SDLC, as recommended by Kitchenham et al [16].

The search strings used were variations of: "reinforcement learning", "RL", "software engineering", "SE", "software development" "software planning", "software analysis", "software design", "software testing", "software deployment", "software maintenance", "development" "planning", "analysis", "design", "testing", "deployment", "maintenance". The digital libraries we queried are listed in Table 1.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

Table 1. List of Digital Libraries.

Digital libraries	URLs
IEEE Xplore	http://www.ieee.org/web/publications/
ACM Digital Library	http://portal.acm.org
Science Direct	http://www.sciencedirect.com/
Springer Link	https://link.springer.com/
Wiley InterScience	https://onlinelibrary.wiley.com/
Elsevier	http://www.elsevier.com
Google Scholar	https://scholar.google.com
DBLP	https://dblp.org

3.3. Inclusion/Exclusion Criteria for Literature.

We defined inclusion and exclusion criteria, which are summarized in Table 2. Included studies consist of original research papers, such as conference papers, journal articles, or accessible preprints, that apply RL techniques to address SDLC phases. Papers must be in English and published between January 2020 and March 2026. We excluded studies not focused on SDLC phases, secondary research such as surveys and meta-analyses, poster abstracts, and duplicate publications.

Table 2. Inclusion and Exclusion Criteria

Inclusion Criteria	
IC1	The paper is written in English.
IC2	The paper is published between January 2020 and March 2026 (inclusive).
IC3	The paper is published/accepted in a peer-reviewed journal, conference or workshop.
IC4	The paper applies reinforcement learning to solve software engineering tasks especially in any SDLC phase (Planning, Analysis, Design, Development, Testing, Deployment, or Maintenance).
IC5	The paper presents original empirical results, case studies, or novel RL-based methods/tools for SE problems.
IC6	The paper provides sufficient technical details on the RL formulation (e.g., state/action space, reward design, algorithm used) to allow reproducibility or clear understanding.
Exclusion Criteria	
EC1	The paper does not apply reinforcement learning (e.g., only uses supervised learning, rule-based methods, or other non-RL paradigms).
EC2	The paper is not focused on software engineering tasks or SDLC phases (e.g., general RL theory, robotics, games without SE relevance).
EC3	The paper is a duplicate publication, poster abstract, demo paper, tool paper without RL evaluation, or lacks sufficient technical depth on the RL approach.

3.4. Snowballing

To ensure the comprehensiveness of our review, we supplemented our primary search by employing snowballing under the guidelines of Wohlin[17]. This involved:

- **Backward Snowballing**: Reviewing the references of the selected papers from prior RL in SE studies to identify earlier works that discuss relevant applications.
- **Forward Snowballing**: Identifying newer studies that have cited the selected papers, ensuring that recent developments in RL for SE were not overlooked.

For each round of snowballing, we applied the inclusion and exclusion criteria to screen the articles. This iterative process was conducted until no new relevant studies were identified, signaling saturation in the search. To maintain consistency and avoid selection bias, two authors independently conducted the snowballing process. Discrepancies in the selection of papers were resolved through discussion and, if necessary, by consulting a third author. This collaborative approach ensured

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

thorough coverage and minimized the risk of omitting key studies. The combination of systematic search and snowballing enabled us to capture a broad spectrum of relevant literature, ensuring that the resulting set of RL applications in SE is both comprehensive and up-to-date.

3.5. Quality Assessment

To ensure the rigor and reliability of the reviewed studies, we conducted a systematic quality assessment on all included papers using the checklist presented in Table 3. The purpose of this assessment was not to exclude studies based solely on subjective quality but to evaluate the strength of empirical evidence, methodological rigor, and clarity of reported contributions, thereby aiding in the contextual interpretation of findings. Two reviewers independently assessed each paper using a three-level scoring scale (Yes/Partially/No). Disagreements were resolved through discussion to reach consensus. The assessment results were used to contextualize findings during synthesis rather than as strict exclusion criteria, ensuring that innovative but early-stage research was not overlooked while accounting for variations in study quality.

Table 3. Quality Checklist of Literature

Clarity of RL Formulation and Description	
BD1	Does the study clearly describe the RL formulation, including state space, action space, reward function, and environment setup?
BD2	Are the RL components relevant and well-justified in the context of the targeted software engineering task or SDLC phase?
Relevance to Software Engineering	
SE1	Is the RL approach applied to specific software engineering tasks, such as automated testing, code generation, fuzzing, or other SDLC phases?
SE2	How well does the RL method address real-world challenges in software engineering (e.g., sparse rewards, high-dimensional states, non-determinism)?
Methodological Rigor	
MR1	Is the RL experimental design well-documented, including algorithm selection, hyperparameter tuning, baselines, and statistical analysis?
MR2	Are the evaluation metrics (e.g., coverage, fault detection rate, efficiency) appropriate, clearly defined, and rigorously applied?
Reproducibility and Availability	
RA1	Are the code, environments, datasets, or artifacts (e.g., RL policies, reward implementations) publicly available or sufficiently described for reproduction?
RA2	Does the study provide enough implementation details (e.g., training setup, episode length, reward scaling) to allow independent replication?
Impact and Contribution to the Field	
IF1	Has the proposed RL method, reward design, or findings been cited or built upon in subsequent RL-SE research?
IF2	Does the study advance the understanding or practical application of reinforcement learning in software engineering?

3.6. Data Analysis

Once we had defined the search time range (2020 Jan. - 2026 Mar.), application domains, publication venues, and languages, and had completed the web scraping, snowballing, and rigorous quality assessment of all candidate papers, we moved to the data analysis and classification phase to answer the research questions.

We extracted the following information from each included study:

- **Bibliographic Information:** Author(s), publication year, title, venue, and citation count.
- **SDLC Phase and Research Focus:** Which of the seven SDLC phases (Planning, Analysis, Design, Development, Testing, Deployment, Maintenance) the work targets, and the specific software engineering task addressed.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

- **RL Formulation:** State space, action space, environment modeling, and the RL algorithm employed (e.g., DQN, PPO, SAC, multi-agent RL).
- **Reward Design and Modeling:** Detailed description of the reward function, including its components (sparse/dense, intrinsic/extrinsic, multi-objective), reward shaping techniques, and any discussion of credit assignment or reward hacking.
- **Experimental Methodology:** Training setup, hyperparameter tuning, baselines, evaluation metrics (e.g., coverage, fault detection rate, efficiency, success rate), and statistical validation methods.
- **Key Results and Impact:** Main findings, performance improvements over traditional methods, practical implications for software engineering workflows, and reported scalability or limitations.
- **Challenges and Limitations:** Any issues reported by the authors regarding sparse rewards, high-dimensional states, sample efficiency, safety, or generalizability.
- **Reproducibility and Availability:** Availability of code, environments, reward implementations, trained policies, or artifacts that enable reproduction.

Data extraction was performed independently by two authors. In cases of discrepancies, discussions were held to reach a consensus, ensuring the accuracy and completeness of the extracted data. This structured extraction enabled us to perform both quantitative trend analysis and qualitative synthesis focused on reward modeling strategies across the software development life cycle.

4. RQ1: What Changes Have Occurred in Reinforcement Learning Research in Software Engineering in Terms of the Number of Published Papers?

This study uses demographic analysis to visualize the distribution characteristics of software engineering studies related to reinforcement learning, focusing on publication year, publication venue, author composition, and type of research contribution. Recent studies [3] indicate a significant upward trajectory in RL-for-SE research, with publications accelerating since 2022 and peaking in 2024 - 2025, with continued activity into March 2026. This growth reflects increasing recognition of RL's potential in addressing complex SE challenges like test generation, code completion, and optimization tasks. The analysis highlights a concentration in software quality assurance, while underscoring opportunities for broader application across the software development lifecycle.

4.1. Publication Trends in Research

Overall Growth Trend. As shown in Figure 2, the publication trends in reinforcement learning applications within software engineering demonstrate a marked increase over the period from 2020 Jan. - 2026 Mar., signaling growing interest and investment in this interdisciplinary field. In 2020, there were 5 publications, which decreased slightly to 4 in 2021. A substantial uptick occurred in 2022 with 11 papers, followed by a sharp rise to 20 in 2023. The momentum continued into 2024 with 21 publications and remained strong at 21 in 2025, with 3 additional papers already recorded in March 2026.

Drivers of Acceleration. This strong growth, particularly evident from 2022 onward, aligns with broader AI advancements [18,19], such as improved deep RL algorithms like PPO and the integration of large language models for code-related tasks. The increase can be attributed to several factors documented in the reviewed literature and supporting AI reports: enhanced computational resources and GPU accessibility, the availability and maturation of open-source RL frameworks like RLlib [20] and Stable-Baselines3 [21], and the demonstrated efficacy of RL in automating SE processes such as test case generation and bug localization. Notably, the sustained high volume in 2024 - 2025 suggests rapid innovation and a shift toward faster dissemination of mature, peer-reviewed results. Overall, this trend underscores RL's maturation from exploratory studies to foundational research, with projections indicating continued expansion as RL addresses emerging challenges in autonomous software development and maintenance.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

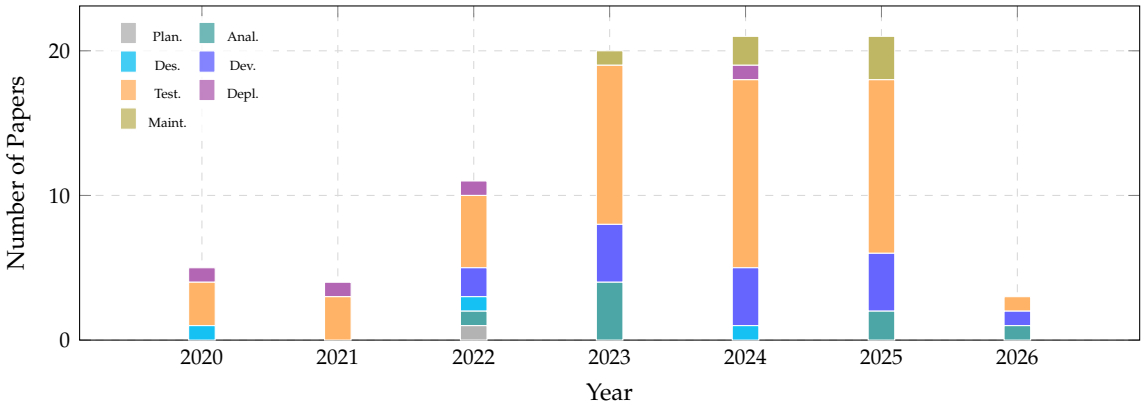

Figure 2. Annual publication counts (Jan. 2020 – Mar. 2026)

4.2. Distribution of Studies in Publication Venues

Venue Distribution Overview. As illustrated in Figure 3, the distribution of RL-for-SE studies across publication venues from 2020 Jan. - 2026 Mar. reveals a blend of top-tier software engineering conferences and journals, alongside prominent AI/ML platforms, reflecting the field's interdisciplinary nature. Leading venues include ICSE with 24.71% of publications, ASE with 14.12%, and ISSTA with 10.59%. Other notable venues comprise ICLR (5.88%), FSE (4.71%), and TSE (4.71%).

Shift Toward AI-Centric Venues. This venue diversity highlights a shift from traditional SE outlets to AI-centric ones, with conferences favoring faster cycles and journals providing deeper empirical validations. High-impact venues like ASE and ICSE dominate due to their rigor. The presence of AI/ML venues such as ICLR, ICML, and ACL further underscores the interdisciplinary nature of the field, pointing to an explosive phase of research in 2024 - 2025, potentially driven by advancements in RL algorithms like PPO and DQN applied to SE domains.

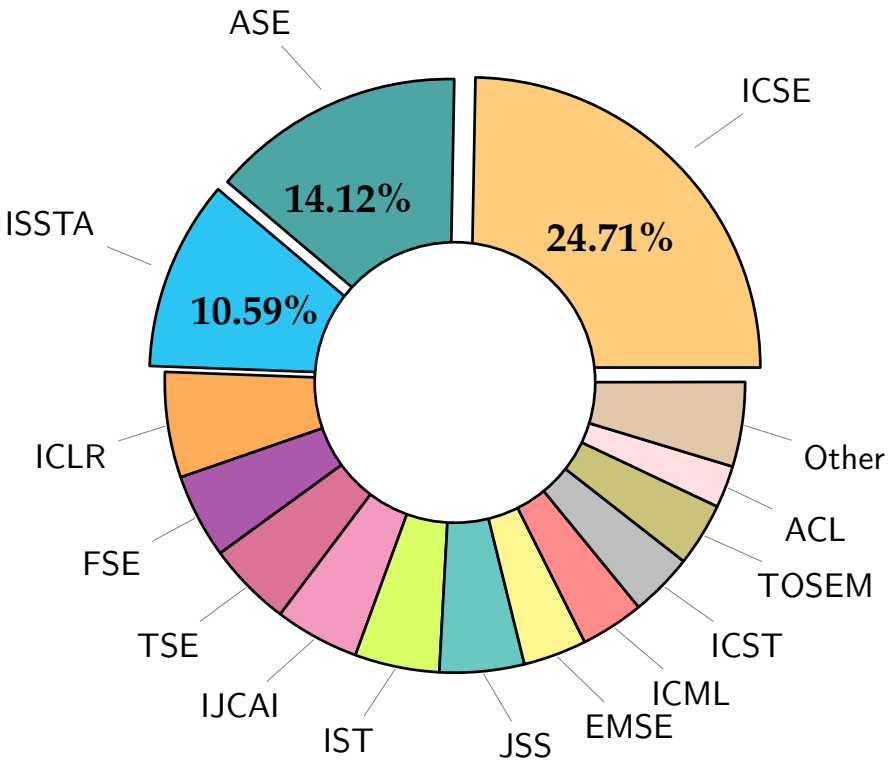

Figure 3. Venue distribution of RL-for-SE papers (Jan. 2020 – Mar. 2026)

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

4.3. Distribution of RL Tasks Each Year

Overall Task Evolution Overview. Reinforcement learning research in the software development life cycle has grown rapidly since 2020, evolving from a handful of exploratory studies concentrated in Testing and Deployment to a mature, diverse body of work spanning all seven SDLC phases. The overall distribution across the 85 surveyed papers is as follows: Planning accounts for 1.18% (1 paper), Analysis for 9.41% (8 papers), Design for 3.53% (3 papers), Development for 17.65% (15 papers), Testing for 56.47% (48 papers), Deployment for 4.71% (4 papers), and Maintenance for 7.06% (6 papers). This heavy concentration in Testing reflects the natural fit of RL for exploration-heavy, reward-driven tasks, while the steady growth in other phases demonstrates RL's expanding role in the full lifecycle.

Early Stage (2020 - 2021): Research in this period was sparse and heavily skewed toward practical validation in dynamic environments. In 2020, studies focused on Design (1 paper), Testing (3 papers), and Deployment (1 paper). By 2021, the emphasis remained on Testing (3 papers) and Deployment (1 paper). These early works primarily explored curiosity-driven and coverage-guided approaches in GUI testing, web testing, and basic scheduling, laying the groundwork for RL as a viable automation tool but with limited coverage of upstream phases such as Planning or Analysis.

Expansion Phase (2022): This year marked the first noticeable broadening of scope. Papers appeared across six phases: Planning (1 paper), Analysis (1 paper), Design (1 paper), Development (2 papers), Testing (5 papers), and Deployment (1 paper). The introduction of code-completion and invariant-learning applications in Development, combined with continued growth in Testing, signaled RL's transition from niche testing aids to more general-purpose tools capable of supporting code synthesis and early-stage optimization.

Acceleration Phase (2023 - 2024): A sharp increase in publication volume occurred, with research exploding in the middle phases. In 2023, contributions included Analysis (4 papers), Development (4 papers), Testing (11 papers), and Maintenance (1 paper). In 2024, the distribution was Design (1 paper), Development (4 papers), Testing (13 papers), Deployment (1 paper), and Maintenance (2 papers). This period saw deep reinforcement learning applied to fine-grained vulnerability detection, multi-step UI testing, code optimization, and self-healing systems, while Deployment and Maintenance began receiving attention for scheduling and issue-resolution tasks. The Testing phase remained dominant but was increasingly complemented by upstream and downstream work.

Sustained Expansion (2025 - March 2026): The field has maintained high productivity levels, with sustained momentum across multiple phases. In 2025, papers covered Analysis (2 papers), Development (4 papers), Testing (12 papers), and Maintenance (3 papers). Early 2026 added Analysis (1 paper), Development (1 paper), and Testing (1 paper). This latest wave features advanced LLM-aligned RL for code generation, preference-based rewards in vulnerability detection, and policy-repair techniques in Maintenance, indicating that RL is now maturing into a comprehensive solution for the entire SDLC rather than remaining a testing-centric technique.

RQ1 Summary

- **Reinforcement learning in software engineering shows a clear upward trend** from 2020 to 2026, with publications increasing from 4-5 per year (2020 - 2021) to around 20 - 21 per year (2023 - 2025), indicating a 4 - 5× growth.
- **Research is heavily concentrated in the Testing phase** (56.47%), while other phases, including Development (17.65%), Analysis (9.41%), and Maintenance (7.06%), expand gradually after 2022; Planning, Design, and Deployment remain limited.
- **Publications are primarily distributed across major software engineering venues** (ASE, ICSE, ISSTA) with increasing presence in AI/ML venues, reflecting growing interdisciplinary engagement.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

5. RQ2: What Types of Software Development Life Cycle Tasks Have Been Facilitated by Reinforcement Learning-Based Methods?

Reinforcement learning has emerged as a powerful paradigm in software engineering, enabling adaptive and optimization-driven approaches across the software development life cycle (SDLC). This research question explores the distribution of RL applications within the seven core SDLC phases - Planning, Analysis, Design, Development, Testing, Deployment, and Maintenance - to identify patterns in task facilitation shown in Table 4, highlight underrepresented areas, and reveal opportunities for future integration. By categorizing existing literature based on these phases, we can discern how RL's ability to learn from interactions and rewards addresses specific challenges, such as automation in testing or optimization in deployment. The analysis is based on the review of 85 studies. Figure 4 shows the distribution of papers by the specific SDLC tasks that they address, and Figure 5 presents all tasks and the corresponding literature.

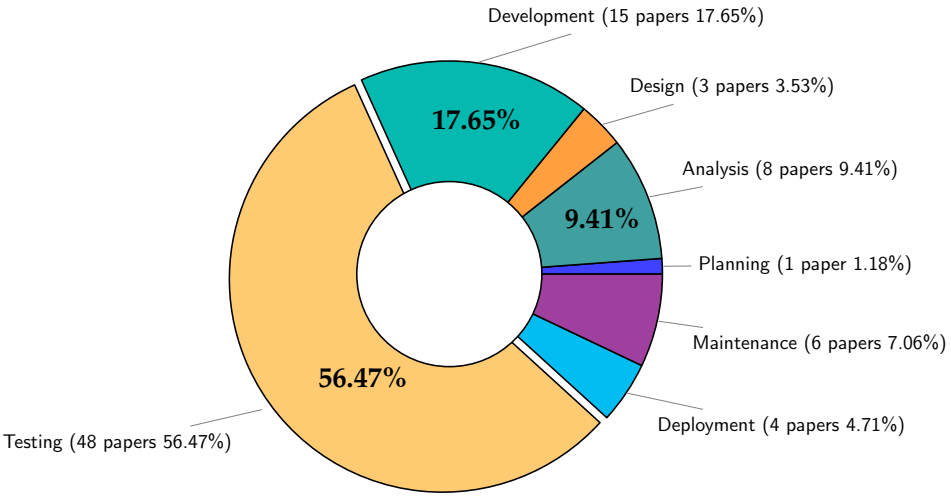

Figure 4. Percentage of papers by SDLC task category

Table 4. Software Development Life Cycle (SDLC) Task Categories

ID	Task	Description
T1	Planning	Defines project goals, scope, resources, timeline, budget, and feasibility. Identifies risks and creates a high-level plan.
T2	Analysis	Gathers and analyzes user & business requirements through stakeholder input. Produces a detailed Software Requirements Specification (SRS).
T3	Design	Creates system architecture, database design, UI/UX, and technical blueprint. Outputs High-Level (HLD) and Low-Level Design (LLD) documents.
T4	Development	Implements features by writing code (front-end, back-end, database, APIs). Includes code reviews and unit testing.
T5	Testing	Performs unit, integration, system, performance, security, and User Acceptance Testing (UAT) to ensure quality and find defects.
T6	Deployment	Releases the tested system to production, including environment setup, data migration, and user training with minimal downtime.
T7	Maintenance	Provides ongoing support: bug fixes, performance tuning, security updates, and feature enhancements after go-live.

Multidimensional Classification Framework. To provide a rigorous and systematic evaluation of the application of reinforcement learning in software engineering, we propose the multidimensional classification framework shown in Table 5. This review not only categorizes existing research according

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

to task type, but also provides an in-depth analysis of each study from the following two core technical dimensions:

1. **Target Domain:** Refers to the specific application scenarios or system environments where RL is applied, such as cloud scheduling or compiler optimization.
2. **Optimization Level:** Denotes at which RL operates, ranging from algorithm-level enhancements to code-level refinements or input-level adaptations the paper is focusing with.

Table 5. Dimensions for each SE Task

ID	Dimension	Examples
1	Target Domain	Cloud resource scheduling, compiler optimization, automated program repair, GUI testing, software verification, code generation, fuzz testing, bug localization, performance tuning, DevOps pipeline optimization
2	Optimization Level	Algorithm-level, architecture-level, code-level, input-level, system-level, requirement-level, etc.

This dual-dimensional analysis highlights RL's versatility in addressing both high-level strategic challenges and low-level operational efficiencies, drawing from a comprehensive review of relevant studies.

5.1. Planning

In the Planning phase, reinforcement learning supports strategic resource allocation and long-term decision-making for project timelines and operations. The Target Domain here focuses on manufacturing and dispatching systems, where RL optimizes planning under uncertainty, such as handling delays in production workflows. For instance, RL methods have been integrated to enhance dispatching in manufacturing environments by learning optimal policies through trial-and-error interactions, allowing for dynamic adjustments that outperform static scheduling heuristics. The Optimization Level operates at the algorithm level, where RL refines planning algorithms to incorporate real-time feedback and long-term rewards. A notable application is in "PDQN" [22] which demonstrates how deep RL variants like Prioritized Deep Q-Networks can effectively manage delayed rewards in planning, leading to more robust and efficient schedules that adapt to unforeseen disruptions in manufacturing pipelines.

5.2. Analysis

This phase leverages RL for data interpretation, pattern recognition, and explanatory modeling in software engineering contexts. Within the target domain, RL is applied to multivariate time series classification, interpretability of RL models, and non-linear reasoning in software systems, often in conjunction with large language models to analyze complex datasets or logs. This domain emphasizes exploratory analysis, such as identifying patterns in time-series data from software metrics or explaining RL decisions in SE tools. RL's ability to learn from sparse or noisy data makes it suitable for these tasks, where traditional machine learning might falter due to lack of supervision. At the optimization level, RL functions at the model level, enhancing the interpretability and reasoning capabilities of analytical frameworks.

For example, Gao et al. [23] integrate RL to guide pattern discovery, improving classification accuracy in SE monitoring tasks by rewarding patterns that align with temporal dependencies. Similarly, Patel et al. [24] and Kenny et al. [25] address the black-box nature of RL in analysis, using prototype-based explanations to make decisions transparent, which is crucial for debugging and validation in

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

software analytics. Hundal et al. [26] warn against assuming uniformity in RL frameworks, showing through empirical analysis how implementation variances affect analytical outcomes in SE testing comparisons, as further explored by Nouwou et al. [27] and Abo et al. [28]. Additionally, Alon et al. [29] combine RL with LLMs for advanced analytical reasoning, while Liu et al. [30] use RL to boost log analysis, ensuring more accurate insights into system behaviors during SE analysis.

5.3. Design

During the Design phase, RL supports architectural and system-level optimizations in software systems. The target domain includes memory placement in heterogeneous computing environments and hierarchical task design for large-scale software systems, where RL automates design explorations that involve trade-offs in performance, efficiency, and reliability. These scenarios benefit from RL's exploratory nature, which can navigate vast design spaces more effectively than exhaustive search methods. The Abstraction Level is primarily at the architecture level, where RL optimizes structural elements such as memory hierarchies or task hierarchies in software designs. Wang et al. [31] employ RL to adaptively learn task hierarchies, resulting in efficient software system designs. Khadka et al. [32] combine evolutionary algorithms with RL to place memory in graph-based representations of software systems, achieving better data locality and reduced latency.

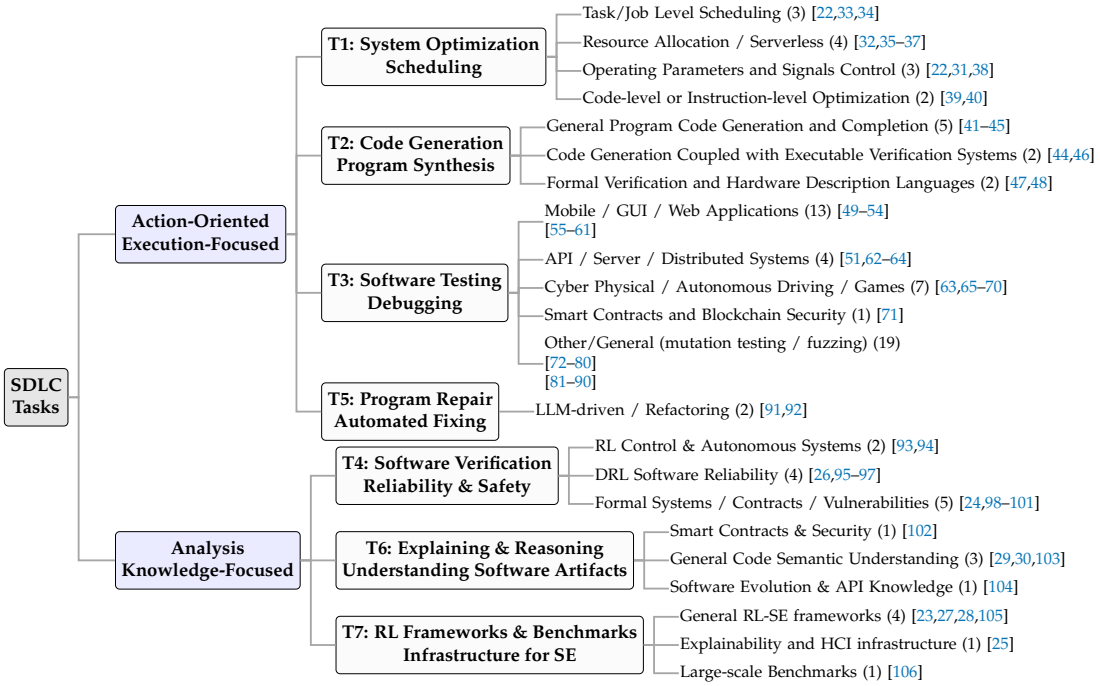

Figure 5. Taxonomy tree of SDLC research topics and literature

5.4. Development

The Development phase sees RL enhancing code generation, optimization, and synthesis tasks. In the target domain, applications span code completion, invariant inference, and refactoring in domains such as Verilog generation [107], contract analysis, and compiler optimization. RL excels here because it naturally models code synthesis and optimization as a Markov Decision Process: each action corresponds to selecting the next token, refactoring step, or optimization transformation, while rewards are derived directly from execution feedback such as compilation success, performance metrics, or functional correctness. This sequential decision-making capability allows to learn policies that generalize across complex codebases far better than purely supervised approaches. The optimization level targets the code level, where RL refines syntactic and semantic elements of code artifacts.

Wang et al. [42] and Li et al. [41] use RL to predict code tokens, incorporating immediate rewards to improve accuracy in large repositories. Teng et al. [47] enhance description language generation through RL fine-tuning, leading to verifiable and efficient code. Invariant learning is addressed by Yu

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

et al. [48], Liu et al. [99], and Wang et al. [100], where RL guides the discovery of invariants through exploration, reducing bugs in developed software. Code optimization features prominently in Lamouri et al. [39], Dou et al. [46], and Tirichine et al. [40], which use compiler feedback to iteratively improve code efficiency. Other works, including Venuto et al. [44], Gorinski et al. [43], Zeng et al. [45], Wang et al. [103], Mindom et al. [105], and Ahmadi et al. [92], integrate RL for broader development aids such as summarization and refactoring, ensuring code quality and maintainability from the outset.

5.5. Testing

Reinforcement learning has emerged as the dominant paradigm in the Testing phase of software engineering, automating test-case generation, prioritization, and fault localization across a wide spectrum of systems. Its primary target domains include mobile applications, web services, cyber-physical systems, autonomous vehicles, and blockchain platforms. Typical tasks range from GUI testing and fuzzing to vulnerability detection in Android, JavaScript, smart contracts, and safety-critical software. The core strength of RL lies in its capacity to intelligently navigate enormous state spaces and learn adaptive policies through interaction with the system under test. By defining rewards based on code coverage, fault revelation, or resource efficiency, RL agents consistently outperform traditional random or model-based methods, especially in domains plagued by combinatorial explosion or non-deterministic behavior.

A prominent line of work exploits curiosity-driven RL to address sparse-reward and exploration challenges. These approaches augment extrinsic rewards with intrinsic signals such as prediction error or state novelty, enabling agents to discover rare edge cases that scripted or random fuzzers routinely miss. Representative examples include Pan et al. [49], whose curiosity-augmented Android fuzzer significantly improves branch coverage by rewarding unexpected app behaviors, and Zheng et al. [50], whose web-testing agent adaptively traverses hyperlinks and forms to expose session-dependent vulnerabilities. Lan et al. [54] further advanced this direction by integrating deep hierarchical state representations, achieving superior fault-detection rates in complex multi-screen GUI applications.

RL-guided fuzzing has equally transformed input-generation techniques. Agents learn mutation policies that maximize long-term coverage rather than relying on purely syntactic or random perturbations. Notable contributions combine RL with large language models for semantically aware mutations (Eom et al. [74]), optimize fuzz inputs for robotic simulation discrepancies (Ren et al. [66]), and craft adversarial transaction sequences to expose reentrancy bugs in smart contracts (Su et al. [71]). In safety-critical domains, Zhang et al. [65] use RL to synthesize sensor-anomaly scenarios, demonstrating practical utility for verifying robustness against real-world perturbations.

Beyond generation, RL excels at test reproduction and prioritization by framing these tasks as sequential decision processes. Agents automatically reconstruct crashes from bug reports or stack traces (Zhang et al. [52] and Huang et al. [55]) and dynamically reorder regression suites according to predicted fault-detection probability (Bagherzadeh et al. [81] and Qian et al. [89]). These methods reduce redundancy, accelerate continuous-integration pipelines, and adapt priorities in response to code evolution, delivering substantial savings in testing time and computational resources.

API and UI testing benefit from RL's multi-step planning capabilities. In RESTful services, agents learn adaptive call sequences that handle stateful responses and uncover authentication or rate-limiting flaws (Kim et al. [62] and Corradini et al. [51]). For GUI testing, bandit-style or image-based RL policies efficiently navigate app hierarchies and cross-platform layouts (Ran et al. [56], Yu et al. [57], and Tang et al. [58]), while robotic-process-automation-inspired agents automate end-to-end user journeys for enterprise regression suites (Puaduraru et al. [59]).

Vulnerability detection has been further strengthened by explainable RL variants and fine-grained analysis. Recent work fine-tunes large language models with RL from human feedback to produce interpretable overflow and injection alerts (Yu et al. [102]) and focuses attention on code snippets to improve precision in smart-contract auditing (Jiang et al. [83]).

Finally, specialized applications extend RL to avionic software [94], formal verification [98], mutation testing [72], compiler bug hunting [77,78], and real-time validation of autonomous driving

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

systems [69]. Hybrid techniques integrating genetic algorithms [93], multi-agent collaboration [53,64], or stability-based oracles [97] effectively tackle domain-specific constraints such as sparse rewards and multi-objective optimization [67,87]. Collectively, these studies demonstrate that RL not only scales to the complexity of modern software but also delivers empirically validated gains in fault coverage, testing efficiency, and release-cycle speed, establishing it as an indispensable tool for handling uncertainty and optimization in contemporary testing workflows.

5.6. Deployment

In the Deployment phase, RL optimizes resource management and scaling in operational environments. The target domain includes heterogeneous clusters, coflow scheduling, and serverless workflows, focusing on runtime efficiency in cloud and streaming systems. RL adapts to varying workloads, learning policies that minimize completion times, such as job completion time, flow completion time in coflow scheduling, and end-to-end latency in serverless workflows. The optimization level is at the system level, tuning configurations for deployment scalability.

Banerjee et al. [34] incorporate domain-specific biases for better scheduling. Sun et al. [33] and Xing et al. [35] use RL for dynamic scaling, while Wang et al. [36] optimize function-as-a-service deployments.

5.7. Maintenance

The Maintenance phase employs RL for issue resolution, self-healing, and performance sustenance. The target domain covers GitHub issue benchmarks, policy repair, and repository maintenance, addressing post-deployment challenges such as bug fixes and API updates. RL enables proactive maintenance by learning from historical data. At the optimization level, it operates at the policy level, repairing or updating RL policies themselves.

Guo et al. [106] benchmark RL for issue handling. Tappler et al. [91] and Yahmed et al. [95] focus on policy repair, while Carmel et al. [96] mitigate issues in RL-driven software. Wang et al. [104] and McIntosh et al. [37] ensure ongoing repository health.

RQ2 Summary

- **Reinforcement learning has been applied across all seven SDLC phases with a focus on several phases.** Testing dominates (>55% of 85 studies), which covers tasks such as GUI/web/-mobile testing, fuzzing, vulnerability detection, and test prioritization.
- Other phases remain less explored, including Development, Analysis, Maintenance, and Deployment, while Planning and Design have minimal coverage.
- The multidimensional classification framework indicates that **RL is primarily applied to action-oriented tasks with explicit reward signals.** Recent work increasingly integrates RL with LLMs and related techniques, suggesting a shift toward broader SDLC coverage.

6. RQ3: Which RL Reward Methods, Design Patterns, and Parameters Have Been Applied?

The integration of reinforcement learning into software engineering represents a fundamental transformation in automation paradigms, shifting from static, heuristic-driven mechanisms toward dynamic, goal-oriented agents capable of navigating the inherent stochasticity and high-dimensional state spaces of the software development lifecycle. At the core of this transformation lies the **reward function**, a critical feedback mechanism that defines the agent's objective landscape and governs policy optimization through structured signals of performance and progress.

Within the context of the seven principal software development life cycle phases identified in RQ2, we classify the reward mechanisms of 85 pioneering studies published between 2020 Jan. - 2026 Mar. according to their **primary reward orientation**. The taxonomy is defined as follows: (1) **Execution-Centric** rewards focus on functional correctness (test pass/fail, compilation success, etc.);

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

(2) **Performance-Driven** rewards target system efficiency (latency, throughput, completion time, etc.); (3) **Intrinsic Curiosity** rewards emphasize exploration and novelty; (4) **Formal Symbolic** rewards rely on verifiers/provers; (5) **Preference-Aligned** rewards incorporate human or semantic preferences; and (6) **Hybrid/Meta** covers multi-objective or learned reward construction. Many studies employ hybrid designs; each paper is assigned to its dominant category only once. Contemporary reward designs increasingly integrate execution semantics, micro-architectural telemetry, structural correctness constraints, and human preference alignment signals, thereby reflecting the escalating complexity of modern software systems and their evaluation criteria.

Table 6 presents the distribution of reward methods across the surveyed literature together with the key parameters involved for each method.

6.1. Execution-Centric Outcome Rewards: The Hard Signals of Correctness

Binary and Proportional Test Passage. In Development and Maintenance phases, reinforcement learning approaches predominantly rely on terminal outcome rewards. Representative frameworks such as CodeRL[108] and SWE-RL[109] are primarily execution based, leveraging repository-level test execution feedback, but often incorporate auxiliary ranking or preference signals to resolve GitHub issues.

In these systems, the agent receives a binary reward, typically 1 if the generated patch passes all tests and fixes the issue, and 0 otherwise. While this formulation ensures objective verifiability grounded in executable semantics, it introduces sparse feedback and delayed credit assignment. To mitigate this limitation, works such as PPO-Coder[110], B-Coder[111], and the compiler test-program generation approach by Chen et al.[77] introduce proportional rewards, where feedback is computed based on the fraction of unit tests successfully passed. This formulation provides smoother optimization signals, enabling agents to distinguish partially correct solutions from entirely incorrect ones.

Building upon this idea, the ACECODER framework[45] further refines execution-based feedback by training a Bradley-Terry reward model on pass-rate comparisons. Similarly, Pytester [88], the boosting input data sequences approach[90], Romdhana et al.[60], Păduraru et al.[59], Ding et al.[79], and Liu et al.[78] employ execution success as rewards, focusing on valid test case generation, sequence coverage, end-to-end functional flows, and fault detection rates.

Constraint-Aware and Hierarchical Reward Designs. Beyond execution outcomes, several frameworks incorporate multi-level constraint-aware rewards to guide agents prior to test verification. SmartCoder-R1[112] formulates the overall reward R as a weighted aggregation of compilation success ($R_{compile}$), security compliance ($R_{security}$), and formatting correctness (R_{format}). The StepCoder framework[46] introduces Function Group Optimization with masked non-executed code segments and the Code Completion Curriculum Strategy. IRCoCo[41] delivers immediate rewards during code completion by detecting dynamic contextual shifts, while Huang et al.[55] reward successful crash reproduction through stack trace validation.

6.2. Intrinsic Curiosity and Exploration-Driven Rewards: Navigating the Search Space

Addressing Reward Sparsity via Intrinsic Signals. In Testing and fuzzing scenarios, the primary objective is to uncover vulnerabilities or maximize execution coverage within expansive state spaces. However, external rewards such as crash discovery are inherently rare events, leading to extreme reward sparsity, a well documented challenge in RL-based software testing[85]. To address this limitation, several approaches incorporate intrinsic motivation mechanisms, wherein agents are rewarded for reaching novel states or executing previously unexplored code paths. Such intrinsic signals transform coverage expansion into a measurable optimization objective, encouraging systematic exploration even in the absence of immediate vulnerability discovery.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

Table 6. Distribution of Reward Method & Parameter

Reward Method	Parameter	Counts	Perc.
Execution-Centric Outcome Rewards	Test pass rate, compilation success, security/format compliance, executed code segments, edit distance, crash reproduction success, mutation isolation precision, fault detection rate	18	**21.2%**
Intrinsic Curiosity & Exploration-Driven Rewards	State/GUI novelty, DFA path novelty, line/branch coverage, RND prediction error, operation order discovery, visual state embedding novelty, hierarchical event priority, multi-step path novelty, FSM sequence shortness, dynamic web page uniqueness, game scenario coverage, adversarial violation discovery	24	**28.2%**
Formal Symbolic Feedback & Proof-Grounded Rewards	Number of counterexamples, remaining proof steps, Hoare triple validity, solver boundary, mutation score, relational invariant satisfaction, runtime check success, Lyapunov stability metric	9	**10.6%**
Preference-Aligned & Semantic Reasoning Rewards	Human preference pairs, pass-rate ranking, VLM screenshot/GUI scores, semantic similarity, hierarchical attention quality, reasoning chain accuracy	9	**10.6%**
Performance-Driven and Metric-Optimized Rewards	Job Completion Time, QoS/latency, execution speedup ratio, memory latency, Mahalanobis distance, repository throughput, kernel execution time, APFD metric	12	**14.1%**
Other / Hybrid / Meta	HTTP status codes, multi-objective mixtures, VLM-generated executable reward code, augmented distributed coverage, sparse CI signals, meta-framework comparisons	13	**15.3%**
	Total	**85**	

Curiosity-Driven in GUI and Web Interaction Testing. Foundational works in this direction include Pan et al. [49], Zheng et al. [50], and the QExplore approach [61]. These systems allocate rewards based on state novelty, transforming unexplored interface components into intrinsic optimization signals. In Pan et al. [49], previously unvisited graphical user interface elements are assigned higher rewards to increase their selection probability. A task-specific parameter, Text, introduces a small positive reward when text input actions are executed, encouraging the agent to prioritize such interactions early in the exploration sequence to unlock deeper application states. Zheng et al. [50] extend a similar exploration strategy to web applications by incrementally constructing a deterministic finite automaton during testing. In this framework, the reward function provides low-level guidance, while the DFA offers high-level structural direction. When the agent becomes trapped-failing to discover novel states-it consults the DFA through a "curiosity timeline" to resume systematic exploration. Yu et al. [57] embed images for state representation, rewarding novel visual states to drive cross-platform exploration. Ran et al. [56] use hierarchical structures to reward event novelty in UI hierarchies. Tang et al. [58] reward multi-step path novelty for improved fault detection in mobile interfaces. Romdhana et al. [80] prioritize rare paths with intrinsic novelty signals for efficient testing in large codebases.

Coverage-Based Intrinsic Reward Designs. Within coverage-guided fuzzing paradigms, reward signals are typically parameterized by line or branch coverage metrics. Eom et al. [74] construct a weighted coverage graph using a term frequency-inverse document frequency scheme, integrating coverage feedback into a large language model-driven mutation framework. This mapping enables the reinforcement learning agent to assign higher rewards to actions that reach rarely explored code regions, effectively steering the LLM-based mutator toward high-risk, low-coverage areas. Jiang et al. [83] reward fine-grained coverage in vulnerability paths for higher recall in code analysis. Turker et al. [82] accelerate FSM exploration with rewards for shorter, novel sequences in model-based testing.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

Exploration in Cyber-Physical and API Testing Systems. The FIGCPS framework by Zhang et al. [65] generates failure-inducing inputs for cyber-physical systems using an integrated reward formula:

$$r_t = in_w \times in_rt + ext_w \times ext_rt \tag{3}$$

where ext_rt is the external reward from the CPS simulator and in_rt is the internal curiosity reward. The internal reward is calculated using Random Network Distillation, defined as the prediction error between a predictor network and a random target network: $in_rt = ||\hat{n}(s_{t+1};\theta) - n(s_{t+1})||^2$. Corradini et al. [51] reward the discovery of effective operation orders that place the API in a testable state, and Kim et al. [62] reward novel API interactions through semantic graph exploration, both apply multi-agent RL to REST API testing. Ferdous et al. [68] drive game scenario exploration with intrinsic novelty for coverage in 3D environments. Doreste et al. [69] reward adversarial scenario discovery for safety violations in simulators. Giamattei et al. [70] extend intrinsic exploration for robustness in cyber-physical systems. Borgarelli et al. [85] augment intrinsic signals for coverage in distributed environments.

6.3. Formal Symbolic Feedback and Proof-Grounded Rewards

Symbolic and Proof-Grounded Reward Signals. For phases requiring absolute reliability, such as Design and Testing, rewards are derived from the outcomes of formal verifiers, SMT solvers, and theorem provers. This represents a symbolic archetype where the reward is logically rigorous.

SMT Solver Synergy and Pruning Rewards. In the inference of loop invariants, the CODE2INV framework by Si et al. [113] models the problem as a multi-step decision process. It provides fine-grained rewards at each step of building the invariant, followed by continuous feedback in the final step based on counterexamples. This counterexample-based smooth reward mechanism allows the agent to iteratively refine its logic until a valid invariant is found that satisfies the automated theorem prover. The LIPuS framework by Yu et al. [48] addresses the sparse reward problem by introducing a two-dimensional reward mechanism. This mechanism enables the RL pruner to recognize the "capability boundary" of SMT solvers and learn pruning heuristics within a few rounds. By synergizing RL with the Z3 SMT solver, Yu et al. [48] can solve more cases than tools like Code2Inv, particularly on non-linear benchmarks where standard rewards fail to guide the agent. Wang et al. [100] reward relational logic satisfaction in formal systems, ensuring invariant correctness through verifier feedback. Kim et al. [101] reward successful runtime checks in Java programs, focusing on contract adherence at unit and system levels.

Reward-Free Verification and Hyperstates. One of the notable reward designs is found in QEDCartographer by Sanchez et al. [98], which automates formal verification in Coq. Instead of explicit rewards, it uses a progress estimation function learned with reinforcement learning. States are modeled as hyperstates, and the target V-value for an obligation is implicitly defined by the number of steps remaining to complete the proof: $V = \gamma^{steps_left}$. Because tactics that finish a proof obligation produce an empty set of obligations, the agent is implicitly rewarded for finishing a proof without needing an explicit reward signal. This design prevents "reward hacking," as the cost of generating an obligation never overcomes the reward of proving it unless it leads to a shorter, more efficient proof. Zhang et al. [97] provide stability-based rewards, using theoretical guarantees to verify control policies in RL software.

Invariant Learning and Contract Reliability. Liu et al. [99] use an offline training phase where the verifier iteratively learns a policy for generating useful contract invariants based on its own verification attempts. The reward is based on the correctness of the generated specification against the Hoare logic rules. Similarly, Tappler et al. [91] use rewards from fuzzing outcomes to update the safety properties of the DRL agent itself, creating a self-healing loop for safety-critical software.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

6.4. Preference-Aligned and Semantic Reasoning Rewards

Preference and Semantic-Level Reward Design. As Large Language Models are increasingly integrated into SE, reward design has evolved to capture human preferences and semantic nuances that execution feedback cannot. This paradigm is crucial for phases like Analysis and Development, like explaining code and generating reasoning traces

Direct Preference Optimization and Ranking. The Smart-LLaMA-DPO framework by Yu et al. [102] utilizes Direct Preference Optimization to detect and explain smart contract vulnerabilities. This approach does not use a traditional scalar reward. Instead, it relies on human experts to generate and validate paired explanations: a high-quality, thorough rationale and a lower-quality counterpart. The loss function encourages the model to increase the probability of the preferred output while decreasing that of the rejected one. This allows the agent to adapt to expert notions of quality and relevance, which is particularly effective for machine-unauditable vulnerabilities that standard static tools fail to explain. Zeng et al. [45] also employ preference alignment by constructing preference pairs based on pass rates over sampled programs to train reward models with Bradley-Terry loss. This approach has led to consistent improvements across competitive benchmarks like BigCodeBench[114] and LiveCodeBench [115]. In a similar vein, aiXcoder-7B-v2 [116] uses a mix of similarity and preference signals for its reinforcement learning policy in code completion. Liu et al.[30] incentivize reasoning chains with preference-aligned rewards for anomaly detection accuracy. Mindom et al. [105] align generalist policies through preference feedback for multi-task SE performance.

Semantic and Similarity-Based Rewards. In phases like Development, rewards are often based on semantic similarity metrics. Representative frameworks such as HAN[103] and TAG[117] optimize for BLEU and ROUGE scores. In subsequent RL-based code generation works, CodeBLEU is frequently used as a more comprehensive metric to capture both syntactic structure and semantic alignment[118]. While these are textual overlap metrics, they serve as proxies for semantic alignment between the code and its natural language description. In code retrieval tasks, such as CoaCor by Yao et al. [119] and QueCos by Wang et al. [120], the Mean Reciprocal Rank is used as a terminal reward, which has been shown to outperform standard similarity-based rewards for ranking quality. Guo et al. [106] introduce a multilingual and multimodal benchmark for GitHub issue resolution, designed to evaluate the ability of large language models to understand and resolve real-world software issues. Wang et al. [103] reward hierarchical semantic attention for improved summarization quality.

Automating Reward Generation with Vision-Language Models. The VLM-CaR framework by Venuto et al. [44] represents a **meta-reward construction** approach. Instead of defining a fixed reward type, a VLM is prompted to generate executable Python programs that define sub-tasks and verify completion based on visual observations. These generated programs are verified on expert/random trajectories before being used to train the RL agent. This method falls under the Hybrid/Meta category in our taxonomy.

6.5. Performance-Driven and Metric-Optimized Rewards

Performance-Centric Reward Formulations. In Deployment-level SE phases, rewards are derived from objective system performance metrics. These signals are typically continuous and tied to resource efficiency. Sun et al. [33] accelerate job completion time by coflow scheduling using a reward tied directly to the minimization of Job Completion Time., by leveraging GNNs to process DAG information, the solutions that optimize the flow of data across heterogeneous clusters. Similarly, the Autoscaler by Xing et al. [35] for streaming jobs uses RL to allocate resources based on Quality of Service requirements. The agent is rewarded for maintaining QoS while minimizing the rescale parallelism value.

In code optimization, the Pearl framework by Lamouri et al. [39] uses execution performance metrics, such as reduced execution time or CPU cycles, as rewards to teach the agent to identify non-obvious code optimizations. Jenkins et al. [22] apply deep reinforcement learning to planning with long delays in manufacturing dispatching, using relative rewards compared to a baseline. Wang et al. [36] reward QoS compliance and latency minimization in hybrid resource allocations. McIntosh

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

et al. [37] reward sustained throughput and reduced latency in repository operations. Tirichine et al. [40] optimize kernel execution time through pass selection rewards.

Multi-Objective and Search-Based Rewards. Testing DNN-enabled systems often requires many-objective reinforcement learning. Agents are rewarded based on multiple parameters, such as path coverage, probability of collision, and system resource stress. In MAEFT by Haq et al. [67], the agent searches for individual discriminatory instances by using the Mahalanobis distance in the feature space as a reward, which encourages the discovery of diverse and rare failures. For memory placement optimization by Khadka et al. [32], evolutionary graph RL is used, where the reward is the inverse of the memory latency, guiding the agent toward efficient memory placements. Bagherzadeh et al. [81] reward average percentage of fault detection metrics for efficient prioritization.

6.6. Comparative Analysis of Reward Parameters and Hyper-Tuning

Cross-Task Parameterization and Hyperparameter Design. The effectiveness of the reward mechanisms discussed in the preceding subsections (e.g., Execution-Centric, Intrinsic Curiosity, Formal Symbolic, Preference-Aligned, and Performance-Driven) is governed by a set of universal and phase-specific parameters. Each of the 85 studies is categorized by its primary reward mechanism, although many approaches incorporate hybrid designs. A synthesis of the reviewed studies reveals common strategies for stabilizing reward signals in software environments.

Discount Factors and Temporal Credit Assignment. The discount factor (γ) is the primary parameter for managing the trade-off between immediate and future rewards. In software engineering, where actions often have long-delayed consequences, γ values are typically high, ranging from 0.95 to 0.99. In the reward-free synthesis of proofs in QEDCartographer by Sanchez et al. [98], γ plays a structural role: it is used to calculate the V-value of hyperstates, where a base case might have a V-value of 0.9 and a recursive case 0.59 . The product of these values, γ^{steps}, serves as the global progress signal, effectively penalizing long action sequences. Yang et al. [87] explicitly handle delayed feedback with adjusted discount factors for CI pipelines.

Integrated Reward Weighting. For many systems, multiple reward sources must be balanced. FIGCPS framework [65] use internal weights (in_w) and external weights (ext_w) to aggregate exploration and goal seeking signals. The evaluation on the Secure Water Treatment (SWAT)[121] testbed, a testbed located in Singapore University of Technology and Design and built for cyber-security research purposes demonstrated that setting both weights to 1.0 provided the necessary balance to discover new attacks in well-tested CPS systems. In many-objective online testing of DNN-enabled software systems, similar weighting parameters are used to balance multiple testing objectives such as path coverage, fault revelation probability, and resource efficiency, preventing the agent from becoming overly focused on one metric at the expense of others [67]. Yang et al. [86] explore weighting schemes for sparse and dense signals in CI contexts.

Reward Scaling and Stabilization Strategies. Software rewards especially performance-based metrics such as makespan or latency, often vary by orders of magnitude across environments and scales[122]. Many frameworks therefore apply batch normalization to reduce variance: rewards are normalized per batch to zero mean and unit variance [123]. The discount factor (γ) is typically set to 0.99. The learning rate (α) is usually chosen in the range 10^{-4} to 10^{-3}. A large negative penalty of -1000 strongly discourages catastrophic outcomes such as system crashes or invalid syntax. Exploration is promoted by a decaying bonus of the form $1/k$, where k is the visit count of a state or action [85]. Finally, a reward threshold of 0.1 is applied when validating external signals in some designs. Collectively, these choices address core RL-SE challenges: stabilizing gradients under extreme reward variance, enabling reliable convergence with PPO/REINFORCE, mitigating policy explosions from sparse signals, and fostering effective exploration in complex software spaces[124].

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

RQ3 Summary

Across 85 RL-for-SE studies, **reward design has evolved from simple binary signals to more complex and multi-component forms**. Intrinsic curiosity-based rewards (28.2%) and execution-centric outcome rewards (21.2%) are the most common, together accounting for nearly half of the studies.

Other reward types include performance-driven metrics (14.1%), preference-aligned and semantic rewards (10.6%), and formal or symbolic rewards (10.6%), with hybrid designs comprising the remainder (15.3%). Common parameter settings include discount factors (0.95 - 0.99) and penalty-based signals for failure cases.

Overall, **reward design trends indicate a shift toward denser and more diverse signals**, with increasing integration of learning-based and structured feedback mechanisms, particularly in Testing and Development phases.

7. RQ4: Which RL Algorithms Have Been Applied?

This research question systematically categorizes the reinforcement learning algorithms employed to enhance various software engineering tasks. It examines their diachronic evolution, adoption patterns over time, and frequency of application across the seven core phases of the software development life cycle: Planning, Analysis, Design, Development, Testing, Deployment, and Maintenance.

The incorporation of RL into SE represents a paradigm shift from traditional rule-based heuristics toward dynamic, autonomous systems capable of adapting to enormous state spaces, non-linear dependencies, and stringent semantic constraints. By framing SE challenges as sequential decision problems in Markov Decision Processes, RL agents observe states, execute actions, and receive rewards to maximize long-term returns such as higher test coverage, faster job completion, or more reliable code. Algorithm selection is driven by phase characteristics: value-based methods suit discrete actions with clear value estimation, while policy-based approaches excel in continuous or high-variance settings. Across the surveyed literature, a broad spectrum emerges, from tabular baselines to deep neural architectures, multi-agent coordination, hierarchical decomposition, and LLM alignment strategies. Earlier works around 2020 leaned on simple Q-Learning for exploratory testing [49,50], while recent studies increasingly adopt sophisticated deep RL variants, PPO-style optimization[110], and reasoning-based methods[125] to handle sparse rewards and complex development and testing domains[3]. Table 7 summarizes the distribution of RL algorithm categories, the specific algorithms utilized, and their predominant application across software engineering tasks throughout the SDLC phases.

Value-Based RL Algorithms. Value-based RL algorithms center on estimating the value of states or state-action pairs, typically through functions like the Q-value, which quantifies the expected future rewards from a given action in a specific state. This estimation guides the agent toward optimal decision-making, making these algorithms particularly well-suited for SE phases involving discrete action spaces, such as selecting test inputs, prioritizing resources, or navigating GUI elements. To overcome the scalability constraints of tabular methods in high-dimensional SE environments, deep value-based algorithms employ neural networks to approximate value functions. The Deep Q-Network (DQN) is the most widely adopted in this category , appearing in 34 papers (40% of the total). It has proved instrumental in large-scale GUI testing [50,54], video-game load testing [63], automatic software refactoring [92], and boosting input sequences for extended finite-state machines. Building on DQN, the Double Deep Q-Network (DDQN) introduces separate networks for action selection and evaluation to mitigate overestimation biases; it is frequently applied in deployment system scheduling [33] and cyber-physical system safety analysis [65]. Truncated Quantile Critics (TQC) further model the full distribution of returns via quantile regression, offering superior sample efficiency and risk-aware learning, as demonstrated in robustness auditing of RL systems [72]. These methods collectively dominate Deployment and Testing phases.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

Table 7. Distribution of Reinforcement Learning Algorithm Categories

RL Algorithm Category	Specific Algorithms	# of Papers	Percentage	Dominant SE Tasks
Value-Based	**Q-Learning** **SARSA** **DQN** **DDQN** **PDQN** **TQC**	34	40%	Deployment, Testing, Planning
Policy/Actor-Critic	**PPO** **A2C/A3C** **SAC** **REINFORCE**	41	48%	Development, Testing, Maintenance
LLM-Alignment	**DPO** **GRPO** **DAPO** **RLEF (Execution-Based)**	24	28%	Development, Analysis, Maintenance
Hybrid/Search-Augmented	**MCTS** **Bayesian RL** **GA-RL** **SB-RLfD** **Fuzzing-guided**	24	28%	Deployment, Testing, Planning
Specialized (MARL, HRL, etc.)	**MARL** **HRL** **Group-RL** **Meta-RL** **Self-Healing**	19	22%	Deployment, Testing, Planning
Advanced Exploration/Reward	**Curiosity (ICM/RND)** **BonusMaxRL** **VLM-CaR** **Perplexity-Based**	30	35%	Testing, Development
Total # of Papers	**85**	*Total # of Papers (with overlaps)*	**172**	

Policy-Based and Actor-Critic RL Algorithms. Policy-based and actor-critic algorithms optimize the probability distribution over actions directly, making them adept at managing continuous action spaces, stochastic policies, and high-variance environments common in Development or adaptive Testing. Actor-critic hybrids further enhance stability by incorporating a critic for value-based variance reduction. Proximal Policy Optimization (PPO) emerges as the leading policy gradient method in this category, valued for its clipped surrogate objective that constrains policy updates and prevents catastrophic shifts in sparse-reward SE scenarios. It is particularly effective in API testing frameworks that sequence requests to expose server flaws [51,62], smart-contract vulnerability detection through reinforced explanations [102], multi-step mobile UI testing, and end-to-end RPA-like testing. The foundational REINFORCE algorithm relies on Monte Carlo estimates of full episode returns; it has been applied in tasks such as learning contract invariants via formal verification feedback [99] and enhancing fine-grained vulnerability detection. Advanced variants include Asynchronous Advantage Actor-Critic (A3C), which leverages parallel actors for decorrelated experiences, and Soft Actor-Critic (SAC), which uses entropy regularization for diverse exploration. These actor-critic methods are prevalent in Development and Testing.

LLM Alignment and Reasoning Algorithms. The fusion of RL with large language models has spawned alignment algorithms that fine-tune generative capabilities for SE-specific goals, using mechanisms like preferences, group comparisons, or execution feedback to ensure outputs are reliable, verifiable, and aligned with software objectives. Direct Preference Optimization (DPO) streamlines alignment by directly learning from pairwise preferences, bypassing explicit reward models; it is harnessed in explainable smart-contract vulnerability detection [102] and incentivizing log analysis in LLMs via reasoning-based RL. Group Relative Policy Optimization (GRPO) computes advantages via intra-group comparisons, while Dynamic Advantage Policy Optimization (DAPO) adapts advantage estimates on-the-fly. Reinforcement Learning from Execution Feedback (RLEF) anchors learning in objective signals from compilers or tests, driving advancements in step-wise code generation [46], automated test synthesis for coding [45], and repository-level completions [42]. These methods prevail in Analysis, Development, and Maintenance phases, with rapid growth in 2024 - 2025.

Multi-Agent, Hierarchical, and Specialized RL Frameworks. For SE problems exhibiting modularity, interdependence, or transfer needs, multi-agent and hierarchical RL frameworks decompose tasks into

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

collaborative or layered structures, facilitating scalable solutions through coordination and knowledge sharing. Multi-Agent RL enables agents to share experiences for collective exploration, boosting efficiency in large-scale web testing [53] and REST API testing with semantic graphs and LLM-driven inputs. Hierarchical RL abstracts decisions via high-level goal setters and low-level executors, excelling in delayed planning in manufacturing [22] and source code summarization. Group-RL promotes heterogeneous knowledge transfer, aiding inductive-biased scheduling in clusters [34]. These specialized frameworks represent 22% of the surveyed works and are key in Deployment and Testing phases.

Hybrid Search-Augmented and Probabilistic RL Algorithms. Hybrid algorithms synergize RL with symbolic search, evolutionary processes, or probabilistic priors to infuse domain knowledge, improve sample efficiency, and tackle combinatorial explosions in SE. Monte Carlo Tree Search (MCTS) directs exploration through simulated rollouts, enhancing model-based testing for state synchronization [75] and RL policy evaluation [73]. Bayesian RL incorporates priors for uncertainty-aware inference, optimizing schedules in heterogeneous clusters [34] and memory placement [32]. Genetic Algorithm-Augmented RL (GA-RL) evolves environments for robust training, while Search-Based RL from Demonstrations (SB-RLfD) synthesizes expert trajectories via fuzzing for policy repair [71,72]. These hybrids lead in Testing and Deployment phases.

Advanced Exploration and Reward Engineering Strategies To combat reward sparsity, a common impediment in SE where successes such as bug detection are rare which is advanced strategies introduce intrinsic motivations and engineered rewards, fostering persistent exploration and denser feedback. Intrinsic motivation techniques reward prediction errors or novelty, driving GUI and web testing to novel states [49,50]. BonusMaxRL uses decaying bonuses for waypoint discovery, aiding protocol verification [72]. VLM-CaR generates code-based rewards from visuals for navigation [44]. Perplexity-based rewards promote fluent code in synthesis [42], and reward augmentation supports testing distributed systems. These techniques, comprising 35% of the surveyed works, are vital for Testing and Development phases. Representative examples include Pan et al. [49] on curiosity-driven Android testing, Zheng et al. [50] on automatic web testing, and Venuto et al. [44] on code-as-reward with vision-language models.

RQ4 Summary

- Across 85 studies, **RL algorithms in SDLC show a diverse distribution.** Policy/actor-critic methods account for 48% (41 papers), followed by value-based methods (40%, 34 papers). Exploration and reward-related strategies appear in 35% of studies, while LLM-aligned, hybrid, and specialized frameworks are also widely used.
- Value-based methods (e.g., DQN variants) are mainly applied in Testing and Deployment, while policy-based methods (e.g., PPO, A3C, SAC) are more common in Development and Maintenance. LLM-aligned approaches have increased in recent years, particularly in Analysis, Development, and Maintenance.
- Overall, the use of RL algorithms has shifted from simple value-based methods to more diverse and hybrid approaches, reflecting broader applicability across SDLC phases.

8. Challenges and Opportunities

Although reinforcement learning has become a dominant paradigm in automated software testing, its practical adoption still faces several critical challenges, while also revealing substantial opportunities for further development. We summarize this section in Table 8.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

Table 8. Challenges and Opportunities

Strengths (Internal) *Technological Capabilities*	**Weaknesses (Internal)** *Technological Limitations*
• **Intelligent State-Space Exploration:** RL agents excel at navigating enormous combinatorial spaces in testing, learning adaptive policies that uncover edge cases far beyond random or model-based methods. • **Curiosity-Driven and Hybrid Policies:** Intrinsic rewards (prediction error, novelty) combined with LLMs, genetic algorithms, or multi-agent systems enable autonomous discovery of rare bugs in dynamic environments. • **Adaptive Test Generation & Prioritization:** Reward functions tied to coverage, fault revelation, and efficiency deliver superior test mutation, crash reproduction, and regression-suite ordering. • **Domain Versatility:** Proven effectiveness across mobile apps, REST APIs, smart-contract auditing, avionic software, and autonomous-driving validation.	• **Sparse, Delayed, and Noisy Rewards.** SE environments often provide only binary signals after long sequences, causing poor credit assignment and reward hacking. • **High-Dimensional and Partially Observable State Spaces.** Large codebases, ASTs, and traces create POMDP-like states; history-dependent policies increase computational cost. • **Scalability and Sample Efficiency.** Real software episodes are expensive; most studies remain limited to small-scale benchmarks. • **Safety, Alignment, and Explainability.** Autonomous agents modifying production code raise risks; interpretability is rarely addressed.
Opportunities (External) *Socio-Technical Environment Potential*	**Challenges (External)** *Socio-Technical Environment Risks*
• **Standardized Benchmark Creation:** Build unified RL-for-SE suites (diverse tasks, verifiable rewards) similar to LLM leaderboards for fair comparison. • **Task-Balance Expansion:** Extend RL into requirements elicitation, architectural design, and long-term maintenance for end-to-end SE automation. • **Real-Time Process Integration:** Embed agents in CI/CD and IDEs for proactive assistance while preserving developer ownership. • **Hybrid Socio-Technical Flywheels:** Use industrial feedback loops and multi-agent systems to refine models and solve sparse-reward issues at scale.	• **Lack of Standardized Benchmarks.** No unified RL-for-SE suite exists for diverse tasks and fair algorithm comparison. • **Task Imbalance.** Most work focuses on testing and code generation; requirements, design, and maintenance remain under-explored. • **Erosion of Safety & Trust.** Over-reliance on autonomous agents risks unintended changes and reduced developer oversight. • **Research & Community Fragmentation.** Without shared benchmarks or safety constraints, progress stays fragmented and biased.

Strengths. Existing research consistently highlights RL's core technological advantages. Agents exhibit exceptional capability in intelligent state-space exploration and adaptive policy learning, particularly in high-dimensional environments such as GUI testing, fuzzing, and code synthesis. The integration of curiosity-driven mechanisms, hybrid architectures like RL+LLM, and sophisticated reward designs further amplifies these strengths, enabling effective bug discovery, test prioritization, and performance optimization across diverse domains.

Weaknesses. Despite these advances, several intrinsic limitations persist. Reward signals in software environments are frequently sparse, delayed, or noisy, complicating credit assignment and increasing the risk of reward hacking. The high-dimensional and partially observable nature of codebases and execution traces leads to substantial computational overhead and poor sample efficiency. Moreover, issues of safety, alignment, and explainability remain largely unaddressed, posing barriers to real-world deployment.

Opportunities. The current landscape suggests several promising directions. Standardized benchmarks are needed to support consistent evaluation and comparison. In addition, RL applications remain concentrated in testing, while other SDLC phases such as requirements engineering, design, and maintenance are less explored. Integrating RL agents into practical development workflows, including CI/CD pipelines and IDE environments, represents an important step toward real-world adoption.

Challenges. Realizing these opportunities requires overcoming several systemic hurdles. The absence of unified benchmarks fragments the community and hinders reproducible comparisons. The pronounced task imbalance, particularly the heavy concentration in testing limits the field's ability to support end-to-end automation. Furthermore, growing reliance on autonomous agents raises critical concerns about safety, trust, and human oversight. Addressing these challenges through collaborative

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

benchmark initiatives, safety aware design principles, and cross phase research will be essential for maturing RL into a foundational technology for autonomous software engineering.

9. Threats to Validity

Internal Validity. Internal validity concerns biases and errors in the review process that could affect the causal inferences drawn from the selected studies. The primary threat arises from potential selection bias during the search and screening phases. Although we employed broad search strings across eight major digital libraries, targeted top-tier venues, and performed iterative backward and forward snowballing until saturation, some relevant papers may have been missed if they used non-standard terminology or appeared in lower-tier venues or non-English outlets.

To mitigate this, two authors independently conducted snowballing rounds, and discrepancies were resolved through discussion. Another threat is data extraction and classification bias. Information on tasks, reward mechanisms, and algorithms was extracted independently by two authors using a structured form. Disagreements were resolved via consensus meetings. However, subjectivity in mapping papers to the seven SE task categories or labeling reward types remains possible. The quality assessment further reduced this risk by contextualizing weaker studies without exclusion. Overall, these multi-reviewer procedures help limit bias, although residual human judgment remains inevitable.

External Validity. External validity addresses the generalizability of our findings beyond the 85 selected studies. Our review is limited to English-language papers published between 2020 Jan. - 2026 Mar. in peer-reviewed venues or arXiv preprints that are accepted by conference. Consequently, results may not generalize to earlier work which is pre-2020, non-English literature, industry reports, or grey literature. The strong concentration of studies in testing and quality assurance venues suggests that findings on reward sparsity and algorithm choice may not fully transfer to underrepresented SE phases such as requirements engineering, maintenance, or deployment.

Additionally, many studies are proof-of-concept evaluations on open-source benchmarks or simulated environments rather than large-scale industrial systems. The rapid growth observed in 2024-2025 and the heavy influence of LLM integration indicate that trends may evolve quickly; thus, our snapshot may not generalize to future RL-for-SE landscapes. By focusing on top-tier and highly cited works plus comprehensive snowballing, we maximized representativeness within the recent academic corpus, but broader generalizability remains constrained by the field's nascency and publication bias toward positive results.

Construct Validity. Construct validity examines whether the operational measures accurately represent the theoretical constructs of "RL applications in SE," "reward modeling," and "algorithm usage." A key threat is that our keyword-based search may have missed papers that apply RL techniques implicitly. We addressed this by including variations of terms and supplementing with snowballing.

The seven-task taxonomy and reward/algorithm classifications were derived inductively from the literature but could impose an artificial structure on heterogeneous studies. For instance, some papers span multiple tasks, leading to possible double-counting or misclassification. The explicit focus on reward design in RQ3 aligns well with the surveyed papers' emphasis on sparse/delayed feedback, yet the construct "reward method" may not fully capture emerging paradigms such as Group Relative Policy Optimization or RL from Verifiable Rewards. To strengthen construct validity, all classifications were cross-validated against the original papers, and explicit mapping examples are provided in the analysis sections.

10. Conclusions and Future Work

This paper presented a systematic survey of recent research on reinforcement learning applications across the software development life cycle. Through a comprehensive review of 85 studies published between 2020 Jan. - 2026 Mar., we analyzed the evolution of RL-driven approaches from publication

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

trends, the distribution of work across the seven SDLC phases, reward design strategies and parameters, and the RL algorithms employed in each phase.

Our analysis shows that reinforcement learning has expanded from primarily testing-focused applications in 2020 - 2021 to work covering all seven SDLC phases by 2025 - 2026. While Testing still dominates (56.47 % of papers), substantial progress has been made in Development (17.65 %), Analysis (9.41 %), Maintenance (7.06 %), Deployment (4.71 %), Design (3.53 %), and Planning (1.18 %). The reviewed literature indicates that reward modeling plays a central role in the effectiveness of RL systems. Reward designs have evolved from simple binary signals to formulations that incorporate intrinsic curiosity, execution semantics, preference alignment, and multi-objective metrics. The choice of reward function influences whether agents learn stable and generalizable policies or encounter reward hacking and training instability.

By mapping existing studies across the full SDLC and centering the analysis on reward modeling, this survey provides a structured overview of how reinforcement learning is applied in software engineering.

Future work should focus on three key directions. First, improved reward modeling techniques are needed, including automated reward generation with vision-language models, better handling of sparse and multi-objective rewards, and standardized reward benchmarks for each SDLC phase. Second, the community would benefit from shared evaluation platforms and reproducible environments that allow fair comparison of different reward strategies and algorithms across phases. Third, deeper integration between reinforcement learning and large language models-particularly through execution-grounded and preference-aligned methods-offers enormous potential for building truly intelligent, lifecycle-spanning software development agents. Addressing these challenges will be essential to move RL from promising prototypes to reliable, production-ready tools that support every stage of modern software development.

References

1. Sutton, R.S. Learning to predict by the methods of temporal differences. *Machine learning* **1988**, *3*, 9–44.
2. Watkins, C.J.; Dayan, P. Q-learning. *Machine learning* **1992**, *8*, 279–292.
3. Wang, D.; You, H.; Zhu, L.; Lin, K.; Chen, Z.; Yang, C.; Yu, J.; Wang, Z.; Chen, J. A Survey of Reinforcement Learning for Software Engineering. *arXiv preprint arXiv:2507.12483* **2025**.
4. Luo, M.; Jain, N.; Singh, J.; Tan, S.; Patel, A.; Wu, Q.; Ariyak, A.; Cai, C.; Venkat, T.; Zhu, S.; et al. DeepSWE: Training a Fully Open-sourced, State-of-the-Art Coding Agent by Scaling RL, 2025. Together AI / Agentica Project. DeepSWE-Preview model also available at https://huggingface.co/agentica-org/DeepSWE-Preview.
5. Wei, Y.; Duchenne, O.; Copet, J.; Carbonneaux, Q.; Zhang, L.; Fried, D.; Synnaeve, G.; Singh, R.; Wang, S.I. SWE-RL: Advancing LLM Reasoning via Reinforcement Learning on Open Software Evolution **2025**. [arXiv:cs.SE/2502.18449]. https://doi.org/10.48550/arXiv.2502.18449.
6. Cao, S.; Li, D.; Zhao, F.; Yuan, S.; Hegde, S.R.; Chen, C.; Ruan, C.; Griggs, T.; Liu, S.; Tang, E.; et al. SkyRL-Agent: Efficient RL Training for Multi-turn LLM Agent **2025**. [arXiv:cs.AI/2511.16108].
7. Masud, M.R.; Wasi, A.T.; Rahman, S.; Parvez, M.R. Reward Engineering for Reinforcement Learning in Software Tasks **2026**. [arXiv:cs.SE/2601.19100].
8. Rayhanul Masud, M.; Toushik Wasi, A.; Rahman, S.; Parvez, M.R. Reward Engineering for Reinforcement Learning in Software Tasks. *arXiv e-prints* **2026**, pp. arXiv–2601.
9. Sutton, R.S.; Barto, A.G.; et al. *Reinforcement learning: An introduction*; Vol. 1, MIT press Cambridge, 1998.
10. Tesauro, G.; et al. Temporal difference learning and TD-Gammon. *Communications of the ACM* **1995**, *38*, 58–68.
11. Ng, A.Y.; Harada, D.; Russell, S. Policy invariance under reward transformations: Theory and application to reward shaping. In Proceedings of the Icml. Citeseer, 1999, Vol. 99, pp. 278–287.
12. Kitchenham, B.; Charters, S.; et al. Guidelines for performing systematic literature reviews in software engineering **2007**.
13. Carrera-Rivera, A.; Ochoa, W.; Larrinaga, F.; Lasa, G. How-to conduct a systematic literature review: A quick guide for computer science research. *MethodsX* **2022**, *9*, 101895.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

14. Khosravi Tabrizi, A.; Ezzati-Jivan, N.; Tetreault, F. An Adaptive Logging System (ALS): Enhancing Software Logging with Reinforcement Learning Techniques. In Proceedings of the Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering, 2024, pp. 37–47.
15. Microsoft. Phases of the Software Development Lifecycle, 2024. Accessed: 2026-03-10.
16. Kitchenham, B.; Brereton, O.P.; Budgen, D.; Turner, M.; Bailey, J.; Linkman, S. Systematic literature reviews in software engineering–a systematic literature review. *Information and software technology* **2009**, *51*, 7–15.
17. Wohlin, C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In Proceedings of the Proceedings of the 18th international conference on evaluation and assessment in software engineering, 2014, pp. 1–10.
18. Ouyang, L.; Wu, J.; Jiang, X.; Almeida, D.; Wainwright, C.; Mishkin, P.; Zhang, C.; Agarwal, S.; Slama, K.; Ray, A.; et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems* **2022**, *35*, 27730–27744.
19. Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* **2017**.
20. Liang, E.; Liaw, R.; Nishihara, R.; Moritz, P.; Fox, R.; Goldberg, K.; Gonzalez, J.; Jordan, M.; Stoica, I. RLlib: Abstractions for distributed reinforcement learning. In Proceedings of the International conference on machine learning. PMLR, 2018, pp. 3053–3062.
21. Raffin, A.; Hill, A.; Gleave, A.; Kanervisto, A.; Ernestus, M.; Dormann, N. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of machine learning research* **2021**, *22*, 1–8.
22. Jenkins, D.C.; Sørensen, R.A.; Singh, V.; Kaminsky, P.; Aswani, A.; Akella, R. PDQN-A Deep Reinforcement Learning Method for Planning with Long Delays: Optimization of Manufacturing Dispatching.
23. Gao, G.; Gao, Q.; Yang, X.; Pajic, M.; Chi, M. A reinforcement learning-informed pattern mining framework for multivariate time series classification. In Proceedings of the 31st International Joint Conference on Artificial Intelligence (IJCAI), 2022.
24. Patel, S.; Jyothi, S.A.; Narodytska, N. CrystalBox: Efficient Model-Agnostic Explanations for Deep RL Controllers **2022**.
25. Kenny, E.M.; Tucker, M.; Shah, J. Towards interpretable deep reinforcement learning with human-friendly prototypes. In Proceedings of the The Eleventh International Conference on Learning Representations, 2023.
26. Hundal, R.S.; Xiao, Y.; Cao, X.; Dong, J.S.; Rigger, M. On the mistaken assumption of interchangeable deep reinforcement learning implementations. *arXiv preprint arXiv:2503.22575* **2025**.
27. Nouwou Mindom, P.S.; Nikanjam, A.; Khomh, F. A comparison of reinforcement learning frameworks for software testing tasks. *Empirical Software Engineering* **2023**, *28*, 111.
28. Abo-eleneen, A.; Palliyali, A.; Catal, C. The role of Reinforcement Learning in software testing. *Information and Software Technology* **2023**, *164*, 107325.
29. Alon, Y.; David, C. Integrating Large Language Models and Reinforcement Learning for Non-Linear Reasoning. *Proceedings of the ACM on Software Engineering* **2025**, *2*, 957–977.
30. Liu, Y.; Chen, Z.; Xu, S.; He, M.; Tao, S.; Meng, W.; Xie, Y.; Han, T.; Zhao, C.; Du, J.; et al. R-Log: Incentivizing Log Analysis Capability in LLMs via Reasoning-based Reinforcement Learning. *arXiv preprint arXiv:2509.25987* **2025**.
31. Wang, Z.; Wang, J.; Zuo, D.; Yunjie, J.; Xia, X.; Ma, Y.; Hao, J.; Yuan, M.; Zhang, Y.; Wu, F. A hierarchical adaptive multi-task reinforcement learning framework for multiplier circuit design. In Proceedings of the Forty-first international conference on machine learning, 2024.
32. Khadka, S.; Aflalo, E.; Marder, M.; Ben-David, A.; Miret, S.; Mannor, S.; Hazan, T.; Tang, H.; Majumdar, S. Optimizing memory placement using evolutionary graph reinforcement learning. *arXiv preprint arXiv:2007.07298* **2020**.
33. Sun, P.; Guo, Z.; Wang, J.; Li, J.; Lan, J.; Hu, Y. Deepweave: Accelerating job completion time with deep reinforcement learning-based coflow scheduling. In Proceedings of the Proceedings of the twenty-ninth international conference on international joint conferences on artificial intelligence, 2021, pp. 3314–3320.
34. Banerjee, S.; Jha, S.; Kalbarczyk, Z.; Iyer, R. Inductive-bias-driven reinforcement learning for efficient schedules in heterogeneous clusters. In Proceedings of the International Conference on Machine Learning. PMLR, 2020, pp. 629–641.
35. Xing, M.; Mao, H.; Xiao, Z. Fast and Fine-grained Autoscaler for Streaming Jobs with Reinforcement Learning. In Proceedings of the IJCAI, 2022, pp. 564–570.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

36. Wang, Y.; Chen, P.; Dou, H.; Zhang, Y.; Yu, G.; He, Z.; Huang, H. Faasconf: QoS-aware hybrid resources configuration for serverless workflows. In Proceedings of the Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 2024, pp. 957–969.
37. McIntosh, S.; Milanesio, L.; Barone, A.; Centkowski, J.; Czech, M.; Ponciroli, F. Using Reinforcement Learning to Sustain the Performance of Version Control Repositories. In Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER). IEEE, 2025, pp. 116–120.
38. Piovarci, M.; Foshey, M.; Erps, T.; Xu, J.; Babaei, V.; Didyk, P.; Matusik, W.; Rusinkiewicz, S.; Bickel, B. Closed-Loop Control of Additive Manufacturing via Reinforcement Learning **2021**.
39. Lamouri, D.R.; Aouadj, I.N.; Kourta, S.; Baghdadi, R. Pearl: Automatic code optimization using deep reinforcement learning. In Proceedings of the Proceedings of the 39th ACM International Conference on Supercomputing, 2025, pp. 959–974.
40. Tirichine, M.; Ameur, N.; Bendib, N.; Aouadj, I.N.; Djad, B.; Bouloudene, R.; Baghdadi, R. A reinforcement learning environment for automatic code optimization in the mlir compiler. *arXiv preprint arXiv:2409.11068* **2024**.
41. Li, B.; Sun, Z.; Huang, T.; Zhang, H.; Wan, Y.; Li, G.; Jin, Z.; Lyu, C. Ircoco: Immediate rewards-guided deep reinforcement learning for code completion. *Proceedings of the ACM on Software Engineering* **2024**, *1*, 182–203.
42. Wang, Y.; Wang, Y.; Guo, D.; Chen, J.; Zhang, R.; Ma, Y.; Zheng, Z. Rlcoder: Reinforcement learning for repository-level code completion. *arXiv preprint arXiv:2407.19487* **2024**.
43. Gorinski, P.; Zimmer, M.; Lampouras, G.; Deik, D.G.X.; Iacobacci, I. Automatic unit test data generation and actor-critic reinforcement learning for code synthesis. In Proceedings of the Findings of the Association for Computational Linguistics: EMNLP 2023, 2023, pp. 370–384.
44. Venuto, D.; Islam, S.N.; Klissarov, M.; Precup, D.; Yang, S.; Anand, A. Code as reward: Empowering reinforcement learning with vlms. *arXiv preprint arXiv:2402.04764* **2024**.
45. Zeng, H.; Jiang, D.; Wang, H.; Nie, P.; Chen, X.; Chen, W. Acecoder: Acing coder rl via automated test-case synthesis. In Proceedings of the Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2025, pp. 12023–12040.
46. Dou, S.; Liu, Y.; Jia, H.; Zhou, E.; Xiong, L.; Shan, J.; Huang, C.; Wang, X.; Fan, X.; Xi, Z.; et al. Stepcoder: improving code generation with reinforcement learning from compiler feedback. In Proceedings of the Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2024, pp. 4571–4585.
47. Teng, F.; Pan, M.; Zhang, X.; He, Z.; Yang, Y.; Chai, X.; Qi, M.; Lu, L.; Yin, J. VERIRL: Boosting the LLM-based Verilog Code Generation via Reinforcement Learning. In Proceedings of the 2025 IEEE/ACM International Conference On Computer Aided Design (ICCAD). IEEE, 2025, pp. 1–9.
48. Yu, S.; Wang, T.; Wang, J. Loop invariant inference through SMT solving enhanced reinforcement learning. In Proceedings of the Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2023, pp. 175–187.
49. Pan, M.; Huang, A.; Wang, G.; Zhang, T.; Li, X. Reinforcement learning based curiosity-driven testing of android applications. In Proceedings of the Proceedings of the 29th ACM SIGSOFT international symposium on software testing and analysis, 2020, pp. 153–164.
50. Zheng, Y.; Liu, Y.; Xie, X.; Liu, Y.; Ma, L.; Hao, J.; Liu, Y. Automatic web testing using curiosity-driven reinforcement learning. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). IEEE, 2021, pp. 423–435.
51. Corradini, D.; Montolli, Z.; Pasqua, M.; Ceccato, M. Deeprest: Automated test case generation for rest apis exploiting deep reinforcement learning. In Proceedings of the Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 2024, pp. 1383–1394.
52. Zhang, Z.; Winn, R.; Zhao, Y.; Yu, T.; Halfond, W.G. Automatically reproducing android bug reports using natural language processing and reinforcement learning. In Proceedings of the Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2023, pp. 411–422.
53. Fan, Y.; Wang, S.; Fei, Z.; Qin, Y.; Li, H.; Liu, Y. Can cooperative multi-agent reinforcement learning boost automatic web testing? an exploratory study. In Proceedings of the Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 2024, pp. 14–26.
54. Lan, Y.; Lu, Y.; Li, Z.; Pan, M.; Yang, W.; Zhang, T.; Li, X. Deeply reinforcing android gui testing with deep reinforcement learning. In Proceedings of the Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024, pp. 1–13.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

55. Huang, Y.; Wang, J.; Liu, Z.; Wang, Y.; Wang, S.; Chen, C.; Hu, Y.; Wang, Q. Crashtranslator: Automatically reproducing mobile application crashes directly from stack trace. In Proceedings of the Proceedings of the 46th ieee/acm international conference on software engineering, 2024, pp. 1–13.
56. Ran, D.; Wang, H.; Wang, W.; Xie, T. Badge: prioritizing UI events with hierarchical multi-armed bandits for automated UI testing. In Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2023, pp. 894–905.
57. Yu, S.; Fang, C.; Li, X.; Ling, Y.; Chen, Z.; Su, Z. Effective, platform-independent gui testing via image embedding and reinforcement learning. *ACM Transactions on Software Engineering and Methodology* **2024**, *33*, 1–27.
58. Tang, X.; Chen, X.; Chen, D.; Zhou, S.; Jiang, W.; Li, Y. TestFlow: Advancing Mobile UI Testing through Multi-Step Reinforcement Learning. In Proceedings of the Proceedings of the 34th ACM SIGSOFT International Symposium on Software Testing and Analysis, 2025, pp. 190–194.
59. Păduraru, C.; Cristea, R.; Stefanescu, A. End-to-end RPA-like testing using reinforcement learning. In Proceedings of the 2024 IEEE Conference on Software Testing, Verification and Validation (ICST). IEEE, 2024, pp. 419–429.
60. Romdhana, A.; Merlo, A.; Ceccato, M.; Tonella, P. Deep reinforcement learning for black-box testing of android apps. *ACM Transactions on Software Engineering and Methodology (TOSEM)* **2022**, *31*, 1–29.
61. Sherin, S.; Muqeet, A.; Khan, M.U.; Iqbal, M.Z. QExplore: An exploration strategy for dynamic web applications using guided search. *Journal of Systems and Software* **2023**, *195*, 111512.
62. Kim, M.; Sinha, S.; Orso, A. Adaptive rest api testing with reinforcement learning. In Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2023, pp. 446–458.
63. Tufano, R.; Masiero, S.; Mastropaolo, A.; Pascarella, L.; Poshyvanyk, D.; Bavota, G. Using pre-trained models to boost code review automation. In Proceedings of the Proceedings of the 44th international conference on software engineering, 2022, pp. 2291–2302.
64. Kim, M.; Stennett, T.; Sinha, S.; Orso, A. A multi-agent approach for rest api testing with semantic graphs and llm-driven inputs. *arXiv preprint arXiv:2411.07098* **2024**.
65. Zhang, S.; Liu, S.; Sun, J.; Chen, Y.; Huang, W.; Liu, J.; Liu, J.; Hao, J. FIGCPS: Effective failure-inducing input generation for cyber-physical systems with deep reinforcement learning. In Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2021, pp. 555–567.
66. Ren, Z.; Li, Y.; Li, X.; Qi, G.; Xuan, J.; Jiang, H. Reinforcement Learning-Based Fuzz Testing for the Gazebo Robotic Simulator. *Proceedings of the ACM on Software Engineering* **2025**, *2*, 1467–1488.
67. Haq, F.U.; Shin, D.; Briand, L.C. Many-objective reinforcement learning for online testing of dnn-enabled systems. In Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2023, pp. 1814–1826.
68. Ferdous, R.; Kifetew, F.; Prandi, D.; Susi, A. Towards agent-based testing of 3d games using reinforcement learning. In Proceedings of the Proceedings of the 37th IEEE/ACM international conference on automated software engineering, 2022, pp. 1–8.
69. Doreste, A.; Biagiola, M.; Tonella, P. Adversarial testing with reinforcement learning: A case study on autonomous driving. In Proceedings of the 2024 IEEE Conference on Software Testing, Verification and Validation (ICST). IEEE, 2024, pp. 293–304.
70. Giamattei, L.; Biagiola, M.; Pietrantuono, R.; Russo, S.; Tonella, P. Reinforcement learning for online testing of autonomous driving systems: a replication and extension study. *Empirical Software Engineering* **2025**, *30*, 19.
71. Su, J.; Dai, H.N.; Zhao, L.; Zheng, Z.; Luo, X. Effectively generating vulnerable transaction sequences in smart contracts with reinforcement learning-guided fuzzing. In Proceedings of the Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022, pp. 1–12.
72. Thomas, D.G.; Biagiola, M.; Humbatova, N.; Wardat, M.; Jahangirova, G.; Rajan, H.; Tonella, P. muPRL: A Mutation Testing Pipeline for Deep Reinforcement Learning based on Real Faults. *arXiv preprint arXiv:2408.15150* **2024**.
73. Tappler, M.; Córdoba, F.C.; Aichernig, B.K.; Könighofer, B. Search-based testing of reinforcement learning. *arXiv preprint arXiv:2205.04887* **2022**.
74. Eom, J.; Jeong, S.; Kwon, T. Fuzzing javascript interpreters with coverage-guided reinforcement learning for llm-based mutation. In Proceedings of the Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2024, pp. 1656–1668.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

75. Türker, U.C.; Hierons, R.M.; Mousavi, M.R.; Tyukin, I.Y. Efficient state synchronisation in model-based testing through reinforcement learning. In Proceedings of the 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2021, pp. 368–380.
76. Reddy, S.; Lemieux, C.; Padhye, R.; Sen, K. Quickly generating diverse valid test inputs with reinforcement learning. In Proceedings of the Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 1410–1421.
77. Chen, J.; Suo, C.; Jiang, J.; Chen, P.; Li, X. Compiler test-program generation via memoized configuration search. In Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2023, pp. 2035–2047.
78. Liu, Y.; Zhu, M.; Dong, J.; Yu, J.; Hao, D. Compiler bug isolation via enhanced test program mutation. In Proceedings of the Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering, 2024, pp. 819–830.
79. Ding, Y.; Zhang, Y.; Yuan, G.; Jiang, S.; Dai, W.; Zhang, Y. Integration test order generation based on reinforcement learning considering class importance. *Journal of Systems and Software* **2023**, *205*, 111823.
80. Romdhana, A.; Ceccato, M.; Merlo, A.; Tonella, P. Ifrit: Focused testing through deep reinforcement learning. In Proceedings of the 2022 IEEE Conference on Software Testing, Verification and Validation (ICST). IEEE Computer Society, 2022, pp. 24–34.
81. Bagherzadeh, M.; Kahani, N.; Briand, L. Reinforcement learning for test case prioritization. *IEEE Transactions on Software Engineering* **2021**, *48*, 2836–2856.
82. Türker, U.C.; Hierons, R.M.; El-Fakih, K.; Mousavi, M.R.; Tyukin, I.Y. Accelerating finite state machine-based testing using reinforcement learning. *IEEE Transactions on Software Engineering* **2024**, *50*, 574–597.
83. Jiang, Y.; Qu, Z.; Treude, C.; Su, X.; Wang, T. Enhancing fine-grained vulnerability detection with reinforcement learning. *IEEE Transactions on Software Engineering* **2025**.
84. Yao, K.; Wang, H.; Qin, C.; Zhu, H.; Wu, Y.; Zhang, L. Carl: Unsupervised code-based adversarial attacks for programming language models via reinforcement learning. *ACM Transactions on Software Engineering and Methodology* **2024**, *34*, 1–32.
85. Borgarelli, A.; Enea, C.; Majumdar, R.; Nagendra, S. Reward Augmentation in Reinforcement Learning for Testing Distributed Systems. *Proceedings of the ACM on Programming Languages* **2024**, *8*, 1928–1954.
86. Yang, Y.; Li, Z.; He, L.; Zhao, R. A systematic study of reward for reinforcement learning based continuous integration testing. *Journal of Systems and Software* **2020**, *170*, 110787.
87. Yang, Y.; Li, Z.; Shang, Y.; Li, Q. Sparse reward for reinforcement learning-based continuous integration testing. *Journal of Software: Evolution and Process* **2023**, *35*, e2409.
88. Takerngsaksiri, W.; Charakorn, R.; Tantithamthavorn, C.; Li, Y.F. Pytester: Deep reinforcement learning for text-to-testcase generation. *Journal of Systems and Software* **2025**, *224*, 112381.
89. Qian, Z.; Yu, Q.; Zhu, H.; Liu, J.; Fu, T. Reinforcement learning for test case prioritization based on LLEed K-means clustering and dynamic priority factor. *Information and Software Technology* **2025**, *179*, 107654.
90. Shu, T.; Wu, C.; Ding, Z. Boosting input data sequences generation for testing EFSM-specified systems using deep reinforcement learning. *Information and Software Technology* **2023**, *155*, 107114.
91. Tappler, M.; Pferscher, A.; Aichernig, B.K.; Könighofer, B. Learning and repair of deep reinforcement learning policies from fuzz-testing data. In Proceedings of the Proceedings of the 46th IEEE/ACM international conference on software engineering, 2024, pp. 1–13.
92. Ahmadi, H.; Ashtiani, M.; Azgomi, M.A.; Saheb-Nassagh, R. A DQN-based agent for automatic software refactoring. *Information and Software Technology* **2022**, *147*, 106893.
93. Liang, L.; Deng, Y.; Morton, K.; Kallinen, V.; James, A.; Seth, A.; Kuantama, E.; Mukhopadhyay, S.; Han, R.; Zheng, X. Garl: Genetic algorithm-augmented reinforcement learning to detect violations in marker-based autonomous landing systems. In Proceedings of the 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE). IEEE Computer Society, 2025, pp. 613–613.
94. Le Tolguenec, P.A.; Rachelson, E.; Besse, Y.; Teichteil-Koenigsbuch, F.; Schneider, N.; Waeselynck, H.; Wilson, D. Exploration-Driven Reinforcement Learning for Avionic System Fault Detection (Experience Paper). In Proceedings of the Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2024, pp. 920–931.
95. Yahmed, A.H.; Bouchoucha, R.; Braiek, H.B.; Khomh, F. An intentional forgetting-driven self-healing method for deep reinforcement learning systems. In Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2023, pp. 1314–1325.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

96. Carmel, O.M.; Katz, G. On Reducing Undesirable Behavior in Deep-Reinforcement-Learning-Based Software. *Proceedings of the ACM on Software Engineering* **2024**, *1*, 1518–1539.
97. Zhang, S.; Song, H.; Wang, Q.; Shen, H.; Pei, Y.; et al. A test oracle for reinforcement learning software based on Lyapunov stability control theory **2025**.
98. Sanchez-Stern, A.; Varghese, A.; Kaufman, Z.; Zhang, D.; Ringer, T.; Brun, Y. QEDCartographer: Automating formal verification using reward-free reinforcement learning **2025**.
99. Liu, J.; Chen, Y.; Tan, B.; Dillig, I.; Feng, Y. Learning contract invariants using reinforcement learning. In Proceedings of the Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022, pp. 1–11.
100. Wang, J.; Wang, C. Learning to synthesize relational invariants. In Proceedings of the Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022, pp. 1–12.
101. Kim, S.; Dutta, S.; Legunsen, O. Valg: A Fast Reinforcement Learning Based Runtime Verification Tool for Java. *ICSE Demo* **2026**.
102. Yu, L.; Huang, Z.; Yuan, H.; Cheng, S.; Yang, L.; Zhang, F.; Shen, C.; Ma, J.; Zhang, J.; Lu, J.; et al. Smart-llama-dpo: Reinforced large language model for explainable smart contract vulnerability detection. *Proceedings of the ACM on Software Engineering* **2025**, *2*, 182–205.
103. Wang, W.; Zhang, Y.; Sui, Y.; Wan, Y.; Zhao, Z.; Wu, J.; Yu, P.S.; Xu, G. Reinforcement-learning-guided source code summarization using hierarchical attention. *IEEE Transactions on software Engineering* **2020**, *48*, 102–119.
104. Wang, S.; Li, Z.; Qian, H.; Yang, C.; Wang, Z.; Shang, M.; Kumar, V.; Tan, S.; Ray, B.; Bhatia, P.; et al. Recode: Robustness evaluation of code generation models. *arXiv preprint arXiv:2212.10264* **2022**.
105. Mindom, P.S.N.; Nikanjam, A.; Khomh, F. Harnessing pre-trained generalist agents for software engineering tasks. *Empirical Software Engineering* **2025**, *30*, 39.
106. Guo, L.; Tao, W.; Jiang, R.; Wang, Y.; Chen, J.; Liu, X.; Ma, Y.; Mao, M.; Zhang, H.; Zheng, Z. Omnigirl: A multilingual and multimodal benchmark for github issue resolution. *Proceedings of the ACM on Software Engineering* **2025**, *2*, 24–46.
107. Thakur, S.; Ahmad, B.; Pearce, H.; Tan, B.; Dolan-Gavitt, B.; Karri, R.; Garg, S. Verigen: A large language model for verilog code generation. *ACM Transactions on Design Automation of Electronic Systems* **2024**, *29*, 1–31.
108. Le, H.; Wang, Y.; Gotmare, A.D.; Savarese, S.; Hoi, S.C.H. CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning. In Proceedings of the NeurIPS, 2022.
109. Wei, Y.; Duchenne, O.; Copet, J.; Carbonneaux, Q.; Zhang, L.; Fried, D.; Synnaeve, G.; Singh, R.; Wang, S.I. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449* **2025**.
110. Shojaee, P.; Jain, A.; Tipirneni, S.; Reddy, C.K. Execution-based code generation using deep reinforcement learning. *arXiv preprint arXiv:2301.13816* **2023**.
111. Yu, Z.; Tao, Y.; Chen, L.; Sun, T.; Yang, H. B-coder: Value-based deep reinforcement learning for program synthesis. In Proceedings of the 12th International Conference on Learning Representations, ICLR 2024, 2024.
112. Yu, L.; Zhang, J.; Wang, X.; Ma, J.; Yang, L.; Zhang, F. Towards Secure and Explainable Smart Contract Generation with Security-Aware Group Relative Policy Optimization. *arXiv preprint arXiv:2509.09942* **2025**.
113. Si, X.; Dai, H.; Raghothaman, M.; Naik, M.; Song, L. Learning loop invariants for program verification. *Advances in Neural Information Processing Systems* **2018**, *31*.
114. Zhuo, T.Y.; Vu, M.C.; Chim, J.; Hu, H.; Yu, W.; Widyasari, R.; Yusuf, I.N.B.; Zhan, H.; He, J.; Paul, I.; et al. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. *arXiv preprint arXiv:2406.15877* **2024**.
115. Jain, N.; Han, K.; Gu, A.; Li, W.D.; Yan, F.; Zhang, T.; Wang, S.; Solar-Lezama, A.; Sen, K.; Stoica, I. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974* **2024**.
116. Li, J.; Zhu, H.; Liu, H.; Shi, X.; Zong, H.; Dong, Y.; Zhang, K.; Jiang, S.; Jin, Z.; Li, G. aixcoder-7b-v2: Training llms to fully utilize the long context in repository-level code completion. *arXiv preprint arXiv:2503.15301* **2025**.
117. Cai, R.; Liang, Z.; Xu, B.; Li, Z.; Hao, Y.; Chen, Y. TAG: Type auxiliary guiding for code comment generation. In Proceedings of the Proceedings of the 58th annual meeting of the association for computational linguistics, 2020, pp. 291–301.
118. Evtikhiev, M.; Bogomolov, E.; Sokolov, Y.; Bryksin, T. Out of the bleu: how should we assess quality of the code generation models? *Journal of Systems and Software* **2023**, *203*, 111741.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 9 October 2026 doi:10.20944/preprints202610.0764.v1

119. Yao, Z.; Peddamail, J.R.; Sun, H. Coacor: Code annotation for code retrieval with reinforcement learning. In Proceedings of the The world wide web conference, 2019, pp. 2203–2214.
120. Wang, C.; Nong, Z.; Gao, C.; Li, Z.; Zeng, J.; Xing, Z.; Liu, Y. Enriching query semantics for code search with reinforcement learning. *Neural Networks* **2022**, *145*, 22–32.
121. Mathur, A.P.; Tippenhauer, N.O. SWaT: A water treatment testbed for research and training on ICS security. In Proceedings of the 2016 international workshop on cyber-physical systems for smart water networks (CySWater). IEEE, 2016, pp. 31–36.
122. Martins, M.S.; Sousa, J.M.; Vieira, S. A systematic review on reinforcement learning for industrial combinatorial optimization problems. *Applied Sciences* **2025**, *15*, 1211.
123. Yuan, D.; Liang, G.; Li, T.; Liu, S. Enhancing queries for code generation with reinforcement learning. *Scientific Reports* **2025**, *15*, 37300.
124. Huang, R. Reinforcement Learning for Safe LLM Code Generation **2025**.
125. Dai, N.; Wu, Z.; Zheng, R.; Wei, Z.; Shi, W.; Jin, X.; Liu, G.; Dun, C.; Huang, L.; Yan, L. Process supervision-guided policy optimization for code generation. *arXiv preprint arXiv:2410.17621* **2024**.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.

© 2026 by the author(s). Distributed under a Creative Commons CC BY license.