Reinforcement Learning from Verifiable Rewards

Kian Kyars

Table of contents

Start Here

Read the book
Download the PDF
View on GitHub

Reinforcement learning from verifiable rewards studies how models can improve by learning from reward signals derived from checkable task outcomes, executable feedback, formal validation, or other reliable forms of verification. This book is a reference on that paradigm. It is not organized around optimizer fashions or a timeline of papers. Its purpose is to explain what kinds of rewards can be made verifiable, what those rewards actually train, where the paradigm has been most successful, and where it breaks.

New to RLVR

Read Chapter 1, Chapter 2, and Chapter 7.

Building Systems

Read Chapter 4, Chapter 5, Chapter 7, and Chapter 9.

Frontier Research

Read Chapter 8, Chapter 10, and Chapter 11.

Flagship Figure

The RLVR pipeline can be read as a stack from objective definition to policy/search updates. Placeholder for the RLVR Verifier Stack figure.

LLM Use

Fortunately, we live in a world where AI slop writing is still *very* intelligible from genuine human text. It is knowing this fact, and also knowing that a textbook is still very much a human-lead endeavor that I write almost all of the sections on my own, or rather use Wispr Flow to dictate them and then edit them. The main contributions of Codex to this project were: - helping me plan out the structure - giving me the initial boilerplate/skeleton scaffold of the textbook itself - creating the diagrams and equations, since this is much more effecient, in particular given my lack of LaTex scripting skills, and is inherently much lower-entropy than writing english, not requiring the same human creativity

Acknowledgments

I shamelessly take inspiration from Nathan Lambert's RLHF book, and I am well aware that his textbook treats the subject of RLVR in quite some detail; notwithstanding, as he notes himself, this particular sub-field of ML is evolving so fast that much of the RLHF book's RLVR content will become outdated, and this book is intended to maintain pace with progress.

Part I

Foundations

1 Introduction

1.1 Chapter Map

- Define RLVR as learning from verifiable reward signals and explain which tasks admit them.
- Explain why RLVR became central to reasoning models and preview the structure of the book.

1.2 What RLVR Is

RLVR is reinforcement learning on tasks where the reward does not need to be guessed from preference comparisons alone because some meaningful part of correctness can be checked directly. Sometimes that check is exact, as in symbolic math or formal proof. Sometimes it is executable, as in code generation with tests. Sometimes it is partial, as in grounded question answering or tool-using agents where only some parts of the trajectory can be reliably scored. The unifying idea is not a specific optimizer. It is the availability of a notion of task success. Once a task can expose useful correctness signals, reinforcement learning can optimize against them, search can exploit them at inference time, and systems can often improve far beyond what static supervised fine-tuning alone would produce.

1.3 Origins of RLVR

In one sense RLVR is the oldest paradigm in reinforcement learning, since it learns from direct reward rather than preference comparison; what is new is its explicit application to language models through verifiers that can check answers, code, proofs, and traces.

I personally reflect back on the advent of reasoning models and reinforcement learning through a strange amnesia of an idea so simple with hindsight, but which took two years after ChatGPT to discover. This assessment, however, is unfair in the sense that the idea to make models think step by step long predates the 2024 reasoning-model wave.[1] The broader prompting paradigm

[1] A useful compressed lineage runs from scratchpads in late 2021, to chain-of-thought prompting in January 2022, to the exact zero-shot prompt "Let's think step by step" in May 2022 (Nye et al. 2021; Wei et al. 2022; Kojima et al. 2022).

emerged across late 2021 and early 2022: scratchpads for intermediate computation appeared first, chain-of-thought prompting then formalized the use of intermediate reasoning traces, and the exact zero-shot prompt "Let's think step by step" was popularized a few months later.

Before the reasoning-model wave of 2024, code generation had already explored reinforcement learning against executable verifiers: CodeRL (July 5, 2022), PPOCoder (January 31, 2023), and RLTF (July 10, 2023) all trained language models using unit tests or execution feedback as objective reward signals.[2]

DeepSeekMath, published on February 5, 2024, was the first major open paper to apply this verifier-driven RL pattern to mathematical reasoning at LLM scale via GRPO.

Things heated up in September 2024, when OpenAI published "Learning to Reason with LLMs", indicating that they had used a train-time and test-time compute strategy to enhance model reasoning through reinforcement learning in math, and coding tasks.[3] The name "Reinforcement Learning with Verifiable Rewards" (RLVR) was coined in the Tulu 3 paper, submitted on November 22, 2024.[4] Finally, there was DeepSeek-R1, which demonstrated the full verifier-driven RL formula for bootstrapping reasoning models.[5] To quote someone describing the atmosphere at Meta after R1 launched, "Engineers are moving frantically to dissect DeepSeek and copy anything and everything we can from it." and according to Fortune, there were war rooms assembled to understand how a Chinese lab with substantially less resources was beating them.[6]

The trend we can extract from this short history is that model improvement increasingly depended on checkable interfaces.

1.4 What Kinds of Tasks Admit Verifiable Rewards

Tasks admit verifiable rewards when they expose an interface that can separate better behavior from worse behavior at acceptable cost. The strongest cases are the familiar ones. Math problems often allow answer checking up to normalization. Code can be run against visible and

[2] CodeRL was submitted on July 5, 2022 and used unit tests and a critic model to guide program synthesis (Le et al. 2022). PPOCoder was submitted on January 31, 2023 and used execution-based feedback with PPO (Shojaee et al. 2023). RLTF was submitted on July 10, 2023 and used online unit-test feedback of multiple granularities for code LLMs (Liu et al. 2023).

[3] OpenAI's writeup states that `o1` performance improved with both more reinforcement learning, which they describe as train-time compute, and more time spent thinking at test time (OpenAI 2024).

[4] DeepSeekMath introduced GRPO and used RL to improve mathematical reasoning in an open model (Shao et al. 2024). Tulu 3 later introduced the name "Reinforcement Learning with Verifiable Rewards (RLVR)" for this broader training pattern (Lambert et al. 2024).

[5] DeepSeek-R1 argues that reasoning abilities can be incentivized through pure reinforcement learning on verifiable tasks such as mathematics, coding competitions, and STEM fields (DeepSeek-AI et al. 2025).

[6] The quoted line was reported as an anonymous Teamblind post summarized by TMTPOST, while the claim that Meta created four "war rooms" was reported by Fortune, citing The Information (TMTPOST Global 2025; Quiroz-Gutierrez 2025).

hidden tests. Formal proof systems can accept or reject proof states under explicit rules. These domains became central not because they exhaust the meaning of reasoning, but because they expose unusually clean signals.

Other tasks are weaker but still useful. Long-context question answering may permit citation checks, evidence matching, or entailment-style grading. Tool-using agents may expose environment transitions, task completion criteria, or execution traces. These signals are often noisier, more expensive, and easier to exploit, but they can still support learning if the reward channel is informative enough.

The practical lesson is that RLVR does not apply uniformly across all tasks. It is strongest where correctness is legible and weakest where the reward channel is sparse, ambiguous, or only loosely coupled to the capability we want.

1.5 Why RLVR Became Central to Reasoning Models

RLVR and reasoning go hand in hand, but they are different. The former is a training paradigm, and the latter is a capability: multi-step breakdown, search, planning, tool use, etc. The marriage between the two occurs because the most successful reasoning domains are exactly the ones with strong verifiers: math, code, proofs, some grounded QA. That combination is rare. It means the same domains that demand search, decomposition, and iterative refinement are also the domains where reinforcement learning has the cleanest chance to work.

This is also why RLVR and reasoning are easy to conflate, and the overlap is large because verifier-friendly domains have been the best places to scale reasoning performance. The result is that some of the most important progress in reasoning models has come from learning against verifiable rewards.

1.6 Verifiable Does Not Mean Complete

Even strong reward signals remain proxies. A math reward may depend on brittle extraction. A code harness may miss behaviors outside the test suite. A proof system may validate a derivation without telling us whether the model's decomposition was insightful or robust. A grounded QA reward may verify some citations without guaranteeing that the answer used evidence faithfully.

That is not a criticism of RLVR so much as a statement of its operating conditions. The important questions are always: what is being checked, what is being missed, how expensive the check is, and how easily the signal can be gamed. Much of the rest of the book is about that gap between a usable reward signal and the fuller competence we actually want.

1.7 What This Book Covers

The next chapters move from the general paradigm to the main reward regimes in practice. Chapters 2 through 4 cover outcome rewards, process rewards, and learned or hybrid verification pipelines. Chapter 5 asks when a check becomes useful learning signal rather than merely a filter. Chapter 6 turns to search and test-time verification, since RLVR in modern systems is inseparable from inference-time compute. Chapters 7 and 8 focus on the main failure modes: reward hacking, proxy misspecification, faithfulness, confidence, and the limits of what verification can certify. Chapters 9 and 10 compare the paradigm across its strongest and most difficult domains. Chapter 11 closes with the open problems.

1.8 References

DeepSeek-AI, Daya Guo, Dejian Yang, et al. 2025. "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning." *arXiv Preprint arXiv:2501.12948*. https://arxiv.org/abs/2501.12948.

Kojima, Takeshi, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. "Large Language Models Are Zero-Shot Reasoners." *arXiv Preprint arXiv:2205.11916*. https://arxiv.org/abs/2205.11916.

Lambert, Nathan, Jacob Morrison, Valentina Pyatkin, et al. 2024. "Tulu 3: Pushing Frontiers in Open Language Model Post-Training." *arXiv Preprint arXiv:2411.15124*. https://arxiv.org/abs/2411.15124.

Le, Hung, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. "CodeRL: Mastering Code Generation Through Pretrained Models and Deep Reinforcement Learning." *arXiv Preprint arXiv:2207.01780*. https://arxiv.org/abs/2207.01780.

Liu, Haochen, Bo Yuan, Yao Hu, et al. 2023. "Reinforcement Learning from Unit Test Feedback." *arXiv Preprint arXiv:2307.04349*. https://arxiv.org/abs/2307.04349.

Nye, Maxwell, Anders Johan Andreassen, Guy Gur-Ari, et al. 2021. "Show Your Work: Scratchpads for Intermediate Computation with Language Models." *arXiv Preprint arXiv:2112.00114*. https://arxiv.org/abs/2112.00114.

OpenAI. 2024. "Learning to Reason with LLMs." September 12. https://openai.com/index/learning-to-reason-with-llms/.

Quiroz-Gutierrez, Marco. 2025. "Meta Is Reportedly Scrambling Multiple 'War Rooms' of Engineers to Figure Out How DeepSeek's AI Is Beating Everyone Else at a Fraction of the

Price." January 27. https://fortune.com/2025/01/27/meta-war-rooms-engineers-deepseek-ai/.

Shao, Zhihong, Peiyi Wang, Qihao Zhu, et al. 2024. "DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models." *arXiv Preprint arXiv:2402.03300*. https://arxiv.org/abs/2402.03300.

Shojaee, Parshin, Mehrdad Li, Arman Cohan, Hannaneh Hajishirzi, Wen-tau Yih, and Sameer Singh. 2023. "PPOCoder: The Effectiveness of Reinforcement Learning for Code Generation." *arXiv Preprint arXiv:2301.13816*. https://arxiv.org/abs/2301.13816.

TMTPOST Global. 2025. "DeepSeek Panic: How Silicon Valley Giants Are Responding to China's AI Disruptor." January 30. https://www.tmtpost.com/7445592.html.

Wei, Jason, Xuezhi Wang, Dale Schuurmans, et al. 2022. "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models." *arXiv Preprint arXiv:2201.11903*. https://arxiv.org/abs/2201.11903.

Part II

Verifier Design

2 Outcome Verifiers

2.1 Chapter Map

- Explain how strong outcome verifiers are built for completed solutions.
- Show why extraction, representation, and hidden brittleness matter more than the apparent simplicity suggests.

2.2 Main Argument

Outcome verifiers are the natural entry point for RLVR because they are operationally simple and often highly scalable. They become useful when the mapping from model output to checked object is stable, unambiguous, and hard to exploit.

This chapter should focus on answer normalization, format design, theorem checking, executable grading, partial credit, and benchmark hygiene. The hard part is often not the final comparison rule but the interface contract that determines what is being compared.

2.3 Canonical Examples

- Exact-match grading for math problems with normalized final answers.
- Code execution against a test suite with hidden tests.
- Formal theorem acceptance in a proof assistant.
- Symbolic evaluation for structured tasks where multiple surface forms represent the same answer.

2.4 Failure Modes

- Checker errors induced by fragile extraction or formatting assumptions.
- Benchmarks that reward shortcuts rather than the intended capability.
- Partial-credit schemes that leak exploitable heuristics.

2.5 What the Verifier Sees

The verifier sees the final artifact: answer string, code file, proof object, or structured output that survives extraction.

2.6 What the Verifier Misses

It misses how the artifact was produced, whether intermediate reasoning was valid, and whether success came from true competence or from exploiting narrow regularities.

2.7 Research Notes

- When is binary scoring enough, and when is graded outcome feedback worth the complexity?
- How should hidden tests be designed to reduce benchmark gaming without drifting away from the task?
- Which extraction conventions are stable across model families?

3 Process Verifiers

3.1 Chapter Map

- Explain when intermediate verification improves credit assignment beyond final-answer rewards alone.
- Show the main risk: rewarding reasoning-shaped traces that correlate weakly with actual competence.

3.2 Main Argument

Process verifiers matter when final-answer rewards are too sparse, too delayed, or too weak to shape behavior reliably. They are not justified by default. A process signal is only helpful if the notion of a "good step" can be defined operationally and checked with enough fidelity.

This chapter should explain how intermediate checks change the learning problem, when they stabilize training, and when they instead create a new proxy that is easier to game than the original task.

3.3 Canonical Examples

- Stepwise checking in mathematical derivations.
- Subgoal validation in formal proofs.
- Intermediate execution checks in long programs or tool-using agents.
- Evidence-selection checks in long-context grounded reasoning.

3.4 Failure Modes

- Rewarding stylistic chains of thought instead of useful intermediate progress.
- Overfitting to the local process checker while harming global task performance.
- Treating process supervision as universally better than outcome supervision.

3.5 What the Verifier Sees

The verifier sees instrumented intermediate states: reasoning steps, subgoal states, partial executions, or cited evidence selections.

3.6 What the Verifier Misses

It misses latent cognition that is not externalized and any beneficial shortcut that bypasses the annotated step structure without harming the final answer.

3.7 Research Notes

- Which tasks admit stable intermediate labels without excessive annotation overhead?
- How do process rewards interact with hidden reasoning or compressed internal computation?
- When should process checks be strict versus advisory?

4 Learned, Programmatic, and Hybrid Verifiers

4.1 Chapter Map

- Explain how to build verifier stacks when a single hard-coded checker is not enough.
- Show the main risk: combining imperfect signals without understanding where their errors compound.

4.2 Main Argument

Many real RLVR systems rely on a stack, not a single verifier. Programmatic checks bring precision, learned judges bring flexibility, and hybrid pipelines make partial verification regimes practical. The design problem is deciding where to trust each layer and how to audit the stack as a whole.

This chapter should center reliability, calibration, appeals, arbitration logic, and how to prevent judge-model weaknesses from becoming the dominant attack surface.

4.3 Canonical Examples

- A programmatic math checker backed by a learned fallback judge for ambiguous outputs.
- A code verifier that combines execution, linting, and judge-model review of unsafe behavior.
- A long-context verifier that mixes citation matching, retrieval consistency, and learned entailment checks.

4.4 Failure Modes

- Silent disagreement between stack components.
- Learned judges inheriting benchmark artifacts or stylistic bias.
- Excessive verifier complexity that makes audits harder than the original task.

4.5 What the Verifier Sees

The stack sees a union of artifacts, traces, retrieved evidence, execution results, and model-judged summaries.

4.6 What the Verifier Misses

It still misses behavior that never enters the instrumented pipeline, along with any systematic blind spots shared by the stack components.

4.7 Research Notes

- When should learned judges be first-class components rather than fallbacks?
- What are the right interfaces for appeals and verifier arbitration?
- How can verifier-stack audits remain tractable as systems grow more layered?

Part III

From Verifiers to Capability

5 Turning Checks into Training Signal

5.1 Chapter Map

- Explain how a checker becomes useful learning signal rather than a brittle scoreboard.
- Keep the focus on signal quality, task shaping, and curriculum rather than optimizer taxonomy.

5.2 Main Argument

The same verifier can be useful or useless depending on how its outputs are turned into signal. Binary versus graded scoring, task selection, filtering, rollout grouping, and curriculum decisions all change the effective optimization landscape before any optimizer-specific choice matters.

This chapter should stay narrowly focused on signal design: how to make checks teachable, how to prevent verifier noise from dominating learning, and how to decide when simple rejection or search-based selection already captures most of the gain.

5.3 Canonical Examples

- Moving from binary pass/fail to graded reward in a math domain with partial structure.
- Filtering tasks to keep the model inside the competence band where signal is informative.
- Using hidden tests or harder variants to keep signal quality from collapsing late in training.

5.4 Failure Modes

- Over-rewarding trivial formatting wins.
- Using a sparse reward regime with no viable path to exploration.
- Smuggling optimizer detail into what should be a chapter about checker design.

5.5 What the Verifier Sees

The verifier sees the same artifacts as before; the new design question is how those outputs are transformed into reward, filtering, or acceptance decisions.

5.6 What the Verifier Misses

It still misses off-policy exploration quality, latent competence, and any capability that the selected signal proxy does not capture.

5.7 Research Notes

- Which signal transformations are robust across domains?
- When is graded reward genuinely better than carefully designed binary reward?
- How can task filtering avoid turning the curriculum into a hidden benchmark hack?

6 Search and Test-Time Verification

6.1 Chapter Map

- Explain what verifiers enable at inference time, not just during training.
- Distinguish immediate gains from search and reranking from gains that are actually amortized into the policy.

6.2 Main Argument

Modern RLVR is inseparable from inference-time compute. Self-consistency, reranking, draft-and-check loops, tool-augmented checking, and structured search all change what the verifier makes possible before the model has fully internalized the behavior.

This chapter should make the training-versus-search boundary explicit. Readers should leave able to separate gains from better policies, better search, and better verifiers.

6.3 Canonical Examples

- Best-of-N math solution selection with an exact answer checker.
- Multi-candidate code generation with unit-test reranking.
- Tool-using agents that query external systems and verify intermediate tool outputs before continuing.

6.4 Failure Modes

- Reporting test-time-search gains as if they were pure policy improvements.
- Ignoring latency and cost when verifier-heavy search is deployed.
- Building search around a weak verifier and amplifying the wrong behavior.

6.5 What the Verifier Sees

The verifier sees multiple candidate outputs, revisions, search nodes, or tool traces rather than a single final sample.

6.6 What the Verifier Misses

It still misses counterfactual search paths not explored and any capability not expressed through the chosen candidate set.

6.7 Research Notes

- Which verifier regimes benefit most from search before training?
- How should evaluations separate amortized capability from search-assisted capability?
- When does search amplify reward hacking rather than competence?

Part IV

Failure Modes

7 Reward Hacking, Proxy Misspecification, and Verifier Robustness

7.1 Chapter Map

- Explain how verifier-driven optimization fails under pressure.
- Treat the checker as an attack surface, not just as an evaluator.

7.2 Main Argument

Any serious RLVR book must treat reward hacking as first-class content. Checker bugs, extraction exploits, benchmark artifacts, weak hidden tests, and biased learned judges all create opportunities for optimization to go in the wrong direction while still looking successful under the nominal reward.

This chapter should teach readers to think adversarially about verifiers: how they are exploited, how those exploits are discovered, and which hardening moves are worth the added complexity.

7.3 Canonical Examples

- Formatting hacks that pass an answer extractor while avoiding the intended reasoning task.
- Unit-test overfitting in code generation.
- Judge-model exploitation through stylistic cues or adversarial phrasing.

7.4 Failure Modes

- Treating benchmark score gains as evidence that the verifier is sound.
- Adding complexity to the stack without improving auditability.
- Leaving hidden tests too close to visible tests.

7.5 What the Verifier Sees

The verifier sees the narrow interface it was designed to score and any auxiliary evidence explicitly exposed to it.

7.6 What the Verifier Misses

It misses behavior outside that interface, including subtle exploit strategies that preserve nominal correctness on the checked slice while breaking the intended task.

7.7 Research Notes

- Which verifier-hardening techniques generalize across domains?
- How should robustness be evaluated before large-scale optimization begins?
- When do learned verifiers create more attack surface than they remove?

8 Faithfulness, Confidence, and What Verification Misses

8.1 Chapter Map

- Explain what important properties remain off-screen even when a verifier is strong.
- Show why verified success does not imply faithful reasoning or calibrated uncertainty.

8.2 Main Argument

RLVR can produce correct answers without guaranteeing faithful reasoning, transparent internal computation, or calibrated confidence. That is not a reason to abandon RLVR. It is a reason to specify the limits of verification clearly and resist overclaiming.

This chapter should give the reader a principled way to talk about what verifiers cannot see, even when they are strong at the task they were built for.

8.3 Canonical Examples

- Correct final answers paired with misleading chain-of-thought traces.
- Self-verification loops that sound confident while remaining wrong.
- Agents that satisfy tool-based checks while remaining brittle under minor environment changes.

8.4 Failure Modes

- Treating visible reasoning as if it were complete access to model cognition.
- Conflating confidence expression with calibrated belief.
- Using process traces as evidence of truth without checking whether they are causally relevant.

8.5 What the Verifier Sees

The verifier sees externalized answers, explanations, traces, and confidence expressions that have been explicitly surfaced.

8.6 What the Verifier Misses

It misses hidden reasoning, latent uncertainty, and any mismatch between narrated reasoning and actual causal computation.

8.7 Research Notes

- Which faithfulness claims are empirically defensible in verifier-heavy systems?
- How should confidence be trained or evaluated when correctness is verifiable but uncertainty is not?
- Can richer verifier stacks reduce blind spots without pretending to eliminate them?

Part V

Domains and Frontiers

9 Canonical Domains: Math, Code, and Formal Proof

9.1 Chapter Map

- Compare how verifier regimes differ across math, code, and formal proof.
- Show why strong checkability does not mean the same reward interface or the same failure modes.

9.2 Domain Overview

Math, code, and formal proof should anchor the book because they expose different strengths and weaknesses of verifiable training. Math offers clean final answers, code offers executable rewards with brittle infrastructure, and proof assistants offer the strongest formal checks but demand more structure and search.

This chapter should compare the verifier interface across the three domains rather than merely survey results.

9.3 Verifier Regime

- Math verifier: A checker centered on answer extraction, symbolic equivalence, or structured derivation validity.
- Code verifier: A checker centered on execution, tests, static constraints, and runtime behavior.
- Formal proof verifier: A checker centered on proof objects and proof-assistant acceptance.

Each regime exposes a different checked object and a different set of bottlenecks. Math emphasizes clean targets and extraction. Code emphasizes execution realism and infrastructure quality. Proof emphasizes decomposition, search, and formal acceptance.

9.4 Canonical Cases

- Final-answer math grading with normalized extraction.
- Unit-test-based code verification with hidden tests and flaky-environment controls.
- Lean or other proof-assistant verification of generated proof steps.

9.5 Comparative Lessons

- Borrowing evaluation intuitions from math and applying them unchanged to code or proof is usually a mistake.
- Executable domains make verifier quality depend heavily on infrastructure rather than only on scoring logic.
- Formal proof exposes the strongest clean verifier but also the sharpest decomposition and search constraints.
- Process verification plays a different role in each domain and should not be ported mechanically.

9.6 Research Notes

- Which lessons from proof systems transfer back to less formal domains?
- How should code verifiers be hardened against flaky or underspecified tests?
- When should math domains use process verification instead of answer-only checks?

10 Long-Context, Multimodal, and Agentic RLVR

10.1 Chapter Map

- Examine how RLVR stretches into long-context, multimodal, and agentic settings.
- Show why frontier tasks remain partially verifiable but become noisier, broader, and easier to misread.

10.2 Domain Overview

This chapter should show how RLVR stretches beyond the cleanest domains without pretending the frontier is neat. Once evidence selection, perception, and environment interaction matter, the verifier often becomes a stack of partial checks rather than a single decisive rule.

The point is to show where verifier-first thinking still works and where it starts to fray.

10.3 Verifier Regime

- Long-context verification: Checking grounded use of evidence over extended context windows.
- Multimodal verification: Checking outputs that depend on visual, auditory, or other non-textual evidence.
- Agentic verification: Checking behavior that unfolds through tool use, environment interaction, and temporal traces.

These settings move the field away from single clean checkers and toward partial, layered, and instrumented verification. The central design question becomes how much of the real task the verifier stack actually captures.

10.4 Canonical Cases

- Citation-grounded long-context question answering.
- Vision-language tasks with verifiable perceptual subgoals.
- Tool-using agents whose traces can be checked for execution validity and grounded progress.

10.5 Comparative Lessons

- Long-context tasks force a distinction between answer correctness and grounded evidence use.
- Multimodal tasks make perceptual ambiguity a first-class verifier problem.
- Agentic settings make temporal traces and environment instrumentation part of the verifier interface.
- Partial verification is often still useful, but it should not be oversold as full evaluation.

10.6 Research Notes

- What is the right unit of verification for agentic trajectories?
- How can long-context tasks separate answer correctness from grounded evidence use?
- Which multimodal checks are robust enough to train against rather than only evaluate with?

11 Open Problems and the Research Agenda

11.1 Chapter Map

- Identify the main bottlenecks preventing RLVR from becoming a mature science.
- End with a real research agenda rather than a paper list or optimizer timeline.

11.2 Main Argument

The book should close by synthesizing its claims into a concrete research program. Stronger process verifiers, more trustworthy learned judges, better separation of reasoning gains from search gains, and better integration with uncertainty, abstention, and safety all remain open.

The final chapter should leave the reader with a map of the next hard questions, not with the impression that RLVR is merely waiting for the next optimizer tweak.

11.3 Canonical Examples

- Tasks where stronger verification would unlock qualitatively different training.
- Domains where judge-model reliability is currently the limiting factor.
- Settings where deployment cost, latency, or audit burden dominate model quality concerns.

11.4 Failure Modes

- Turning "open problems" into generic benchmark wishlist items.
- Treating all verifier weaknesses as annotation or scale problems.
- Ignoring deployment constraints in favor of purely academic task formulations.

11.5 What the Verifier Sees

The verifier sees whatever the current interface and instrumentation permit, along with the limits of those choices.

11.6 What the Verifier Misses

It misses the capabilities that would only become legible under richer interfaces, stronger audits, or entirely new verification regimes.

11.7 Research Notes

- Which new verifier classes would most change the field?
- How should RLVR systems report uncertainty about the verifier itself?
- What would a mature verifier-evaluation standard look like?

A Minimal RL and Post-Training Background

A.1 Purpose

This appendix should provide the smallest amount of RL and post-training context needed to read the main text without turning the book into a general RL manual.

A.2 Include

- Trajectories, returns, and policy improvement at a high level.
- KL regularization and why post-training stacks use it.
- The difference between rejection, search, and policy optimization.
- Enough terminology to decode papers without shifting the book's focus away from verifiers.

A.3 Exclude

- Full derivations that belong in a standard RL textbook.
- Optimizer-by-optimizer historical detail unless a verifier-facing concept depends on it.

A.4 Objective, States, and Rewards

At its core, reinforcement learning optimizes the expected discounted return of trajectories in a Markov decision process:

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, R, \gamma).$$

For a policy π, the objective is:

$$J(\pi) = \mathbb{E}_{\tau \sim \pi}\left[\sum_{t=0}^{T-1} \gamma^t r(s_t, a_t)\right].$$

where $\tau = (s_0, a_0, s_1, a_1, \dots)$.

For **single-turn LLM inference/training with one prompt-to-completion interaction**, this collapses to a contextual bandit view: one initial state s_0 (the prompt), one action a (the sampled completion), and then termination:

$$s_0 = x, \quad a = y, \quad s_1 = \text{terminal}, \quad r = r_\phi(x, y).$$

So the objective becomes:

$$J(\pi_\theta) = \mathbb{E}_{x \sim p_{\text{data}}} \left[\mathbb{E}_{y \sim \pi_\theta(\cdot|x)} \, r_\phi(x, y) \right].$$

For **multi-turn settings** we retain the full trajectory form with state as dialogue/context history:

$$s_t = (x, y_{<t}), \qquad a_t = y_t, \qquad s_{t+1} = (x, y_{\le t}).$$

The trajectory distribution factorizes as:

$$\pi_\theta(y_{1:T} \mid x) = \prod_{t=1}^{T} \pi_\theta(y_t \mid x, y_{<t}),$$

and the return is:

$$J(\pi_\theta) = \mathbb{E}_{x \sim p_{\text{data}}} \left[\mathbb{E}_{y_{1:T} \sim \pi_\theta(\cdot|x)} \left[\sum_{t=1}^{T} \gamma^{t-1} r_t(s_t, y_t) \right] \right].$$

In many verifier-driven setups, (r_t 0) for (t<T) and a scalar terminal reward (r_T=R_ (x,y_{1:T})) carries the verification signal from the environment.

In this book, we use (x) for prompts and (y) for generated outputs (or turn-level outputs), and we write the verifier or environment as a score function (R_) or (r_) that maps prompts and completions, or full trajectories, to reward.

B Benchmarks, Evals, and Contamination

B.1 Purpose

Collect evaluation hygiene issues that would otherwise interrupt the flow of the main chapters.

B.2 Include

- Leakage and train-test overlap.
- Extraction mismatches between benchmark and verifier.
- Hidden tests and benchmark saturation.
- Reporting pitfalls such as pass@k misuse or search-assisted evaluation drift.

B.3 Working Note

This appendix should become the reference location for benchmark caveats so the main text can stay focused on verifier concepts.

C Practical Verifier Design Checklist

C.1 Purpose

This appendix should read like a field manual that can be pasted into internal docs or shared directly in lab discussions.

C.2 Core Terms

- **Verifiable reward**: A reward signal derived from an outcome, execution, proof state, trace, or other artifact that can be checked with reasonable reliability.
- **Verifier**: The mechanism that performs that check, whether symbolic, executable, formal, learned, or hybrid.
- **Reward signal**: The scalar or graded feedback that learning actually sees after verification.
- **Interface**: The part of the task exposed to checking, such as a final answer, a program, a proof state, a citation set, or an environment trajectory.
- **Outcome verifier**: A checker that scores a completed solution rather than its intermediate steps.
- **Process verifier**: A checker that scores intermediate reasoning, subgoals, or partial traces.
- **Programmatic verifier**: A rule-based checker implemented through deterministic logic, execution, or formal constraints.
- **Learned verifier**: A model-based judge that predicts correctness, quality, or consistency.
- **Verifier stack**: A layered pipeline that combines multiple checks before producing a reward or decision.
- **Signal quality**: How informative, stable, and hard to game the reward is for the capability of interest.
- **Faithfulness**: The extent to which an externalized explanation tracks the causal basis of the model's answer.
- **Calibration**: The relationship between expressed confidence and actual correctness.

C.3 Checklist

1. What exact object is being verified?
2. What evidence does the verifier actually consume?
3. Which important properties remain off-screen?
4. Where are the obvious attack surfaces?
5. Which failures are silent rather than visible?
6. How will robustness be audited before large-scale optimization?
7. What deployment constraints shape the acceptable verifier stack?