Reinforcement Learning for Safe LLM Code Generation

Roy Huang

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-123
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-123.html

May 19, 2025

Acknowledgement

I would like to acknowledge and thank the entire rLLM team, in particular, Michael Luo and Sijun Tan, for being supportive, responsive, and helpful mentors and introducing me to RL training on LLMs and agents. I would also like to thank Prof. Joseph E. Gonzalez for the supportive advising, guiding me along my journey. Most importantly, I would like to thank my parents for supporting me through my life and getting me to where I am today. I could not have made it without their love and encouragement.

Reinforcement Learning for Safe LLM Code Generation

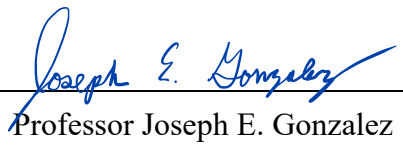
by Yu Fei Huang

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:



Professor Joseph E. Gonzalez
Research Advisor

5/15/2025

(Date)

* * * * * *

Professor Raluca Ada Popa
Second Reader

5/18/2025

(Date)

Reinforcement Learning for Safe LLM Code Generation[1]

by

Yu Fei Huang

A thesis submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Electrical Engineering and Computer Sciences

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Joseph E. Gonzalez, Chair
Associate Professor Raluca Ada Popa

Spring 2025

[1]This thesis is adapted from **GoEX: Perspectives and Designs Towards a Runtime for Autonomous LLM Applications** [26] and **DeepCoder: A Fully Open-Source 14B Coder at O3-mini Level** [17]. It is recommended to cite these papers over this report.

Reinforcement Learning for Safe LLM Code Generation[1]

[1]This thesis is adapted from **GoEX: Perspectives and Designs Towards a Runtime for Autonomous LLM Applications** [26] and **DeepCoder: A Fully Open-Source 14B Coder at O3-mini Level** [17]. It is recommended to cite these papers over this report.

Abstract

Reinforcement Learning for Safe LLM Code Generation[2]

by

Yu Fei Huang

Master of Science in Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Joseph E. Gonzalez, Chair

Reinforcement learning (RL) has become a primary technique for aligning Large Language Models (LLMs) with complex reasoning objectives, yet convergence is fragile when reward signals are noisy or exploitable. This thesis presents rLLM—an open-source, Ray-based RL framework that utilizes an improved Group-Relative Policy Optimization (GRPO+) with veRL modified with asynchronous pipelined sampling, and iterative context lengthening. Using rLLM we trained Deepcoder-14B, a 14-billion-parameter code-reasoning model that attains 60.6 % Pass@1 on LiveCodeBench, a 1936 Codeforces rating, and 92.6 % Pass@1 on HumanEval+, matching OpenAI's proprietary o3-mini (low) and o1 on these benchmarks.

We show that such performance hinges on an airtight sandboxed execution environment that safeguards reward integrity. To that end we take inspiration from GoEx, a post-facto-validated runtime that envelopes every REST call, database mutation, and file operation in deterministic Undo and blast-radius-bounded confinement semantics. The airtight environments which rLLM consumes directly to compute rewards using, eliminating reward hacking.

The findings underscore that the proposed GRPO+ modification significantly enhances training convergence compared to existing widely-adopted algorithms such as GRPO and DAPO. Furthermore, the asynchronous pipelining mechanism incorporated into veRL substantially optimizes the training infrastructure, enabling efficient scalability. Ultimately, by integrating these advancements within a meticulously secure environment, this thesis delivers a comprehensive RL framework that reliably aligns LLMs with sophisticated reasoning objectives, paving the way for future research into robust and scalable reinforcement learning systems.

[2]This thesis is adapted from **GoEX: Perspectives and Designs Towards a Runtime for Autonomous LLM Applications** [26] and **DeepCoder: A Fully Open-Source 14B Coder at O3-mini Level** [17]. It is recommended to cite these papers over this report.

To my parents, advisor, and research collaborators

I would like to acknowledge and thank the entire rLLM team, in particular, Michael Luo and Sijun Tan, for being supportive, responsive, and helpful mentors and introducing me to RL training on LLMs and agents. I would also like to thank Prof. Joseph E. Gonzalez for the supportive advising, guiding me along my journey. Most importantly, I would like to thank my parents for supporting me through my life and getting me to where I am today. I could not have made it without their love and encouragement.

Contents

Contents **ii**

List of Figures **iv**

List of Tables **vi**

1 Introduction **1**
1.1 Background and Motivation 1
1.2 rLLM Framework Overview 2

2 Related Work **4**
2.1 Reinforcement Learning for Language-Model Alignment 4
2.2 Distributed Frameworks and Systems Infrastructure 5
2.3 Secure Execution Environments and Reward Integrity 6

3 GoEX: Execution Runtime for LLMs **8**
3.1 Designing a Runtime for LLM Execution 8
3.2 Reversibility and Damage Confinement 8
3.3 Symbolic Credentials and Sandboxed Execution 9
3.4 Credential Storage and Access Control 9
3.5 System Design Components 9

4 rLLM: RL Training for LLM Reasoning **17**
4.1 Problem Statement 17
4.2 rLLM Framework 18

5 rLLM Experiment: Deepcoder-14B **24**
5.1 Dataset Curation Strategy 24
5.2 Code Sandbox Environment for Reward Computation 25
5.3 Reward Function Design 26
5.4 Evaluation Results 26
5.5 End-to-end Performance 27

6 Conclusion **29**

Bibliography **31**

A Codeforces Evaluation **35**

List of Figures

3.1 GoEX's runtime for executing RESTful API calls. Upon receiving the user's prompt, GoEX presents two alternatives. First, an LLM can be prompted to come up with the (`Action`, `Undo-Action`) pair. Second, the application developer can provide tuples of actions and their corresponding undo-actions (function calls) from which the LLM can pick amongst. 10
3.2 Runtime for executing actions on a database. We present two techniques to determine if a proposed action can be undone. On the left, for non-transactional databases like MongoDB, and for flexibility, we prompt the LLM to generate (`Action`, `Undo-Action`, `test-bed`) tuples, which we then evaluate in a isolated container to catch any false (`Action`, `Undo-Action`) pairs. On the right, we can provide a deterministic undo with guarantees by employing the transaction semantics of databases. 13
3.3 Runtime for executing actions on a filesystem. GoEX presents two abstractions. On the left, the LLM is prompted to come up with an (`Action`, `Undo-Action`, `test-bed`) which GoEX evaluates in a isolated container to catch any false (`Action`, `Undo-Action`) pairs. On the right presents deterministic guarantees by using versioning control system like Git or Git LFS. 15

4.1 Average training reward between GRPO+ and GRPO for the 16K run. GRPO's reward curve eventually collapses. GRPO+'s curve is stable due to **Clip High**. 18
4.2 Due to **overlong filtering**, GRPO+'s response length grows steadily over time. 19
4.3 **Clip High** and **No Entropy Loss** ensures that GRPO+'s token-level entropy does not collapse and encourages sufficient exploration. 19
4.4 DeepCoder's average response length and training rewards as training progresses. Average response length increases from 8K→17.5K context length. 20
4.5 Verl's PPO/GRPO training pipeline. Every RL iteration cycles through sampling, reward function calculation and training. Sampling is the bottleneck; training speed is bounded by straggler samplers that generate long sequences. . 21
4.6 Minibatch Pipelining. Samplers and trainers operate in separate worker groups. As samplers complete and release mini-batches (for PPO/GRPO), trainer workers process them asynchronously. At the end of an iteration, trainers broadcast their weights to samplers. 22

4.7 One-Off Pipelining. Samplers generate a batch one iteration ahead, while trainers update gradients using the previous iteration's data. Second, reward function calculation is interleaved with sampling. This approach does not introduce asynchronous off-policy samples to GRPO/PPO's on-policy algorithm. 23

5.1 One-off pipelining fully masks away trainer and reward computation times, reducing training times by 1.4x for math and 2x for coding. 27

List of Tables

5.1 Model Performance on Coding and Math Benchmarks 27

Acknowledgments

I would like to acknowledge and thank the entire rLLM team, in particular, Michael Luo and Sijun Tan, for being supportive, responsive, and helpful mentors and introducing me to RL training on LLMs and agents. I would also like to thank Prof. Joseph E. Gonzalez for the supportive advising, guiding me along my journey. Most importantly, I would like to thank my parents for supporting me through my life and getting me to where I am today. I could not have made it without their love and encouragement.

Chapter 1

Introduction

Large Language Models (LLMs) have advanced from sequence-to-sequence autoregressors into agents capable of multi-step reasoning, tool calling, and code synthesis. Supervised pre-training supplies fluent linguistic priors, yet it is reinforcement learning (RL) that aligns those priors with task-level objectives such as passing unit-test suites or developing emergent reasoning patterns. Optimizing an LLM policy π_θ over long, sparse reward trajectories, however, remains brittle: credit-assignment noise grows quadratically with sequence length, and poorly instrumented environments invite reward hacking, where policies learn spurious strategies that inflate the scalar return while degrading true utility.

This thesis addresses these challenges by proposing rLLM, a purpose-built RL framework that couples a novel Group-Relative Policy Optimization Plus (GRPO+) algorithm based on prior works with GRPO and DAPO with an asynchronous, Ray-orchestrated sampling pipeline. rLLM's design goal is two-fold: (i) sustain high-throughput gradient updates on clusters of thousands of GPUs; and (ii) preserve reward integrity through airtight execution sandboxes inspired by the GoEx post-facto validation runtime. The framework is validated by training Deepcoder-14B, a 14-billion-parameter code-reasoning model that matches the performance of proprietary systems while remaining fully open source.

1.1 Background and Motivation

The alignment of Large Language Models (LLMs) has progressed from supervised finetuning (SFT) to full reinforcement-learning pipelines that optimize a policy over long, task-level roll-outs. Early RL with human feedback (RLHF) systems adopted Proximal Policy Optimization (PPO) and its KL-constrained variants, but the high variance of long-horizon credit assignment soon motivated Group-Relative Policy Optimization (GRPO), which measures advantages against peer trajectories sampled from the same prompt group, markedly improving stability on reasoning tasks. Subsequent work such as DAPO added dynamic sampling and decoupled clipping to push large-scale training beyond 30 B parameters. Despite these algorithmic advances, convergence is still brittle whenever reward channels leak

noise or are exploitable. Studies on reward hacking show that agents readily discover loopholes—fabricating logs, short-circuiting unit tests, or corrupting state—to inflate nominal returns while degrading true task success.

Scaling RL to frontier-sized models therefore demands system innovations as well. Synchronous actor–learner loops stall on the longest rollout, under-utilising expensive accelerators; industrial solutions now favour asynchronous pipelines built atop Ray's distributed execution engine, which offers elastic, fault-tolerant placement of both actors and learners. Libraries such as veRL expose lightweight RPC interfaces for high-throughput sampling and have become a de-facto substrate for open-source RLHF research. Yet throughput alone is insufficient: long-context Optimization (32 k–64 k tokens) multiplies gradient noise and memory pressure, motivating iterative context lengthening curricula that grow windows only after variance plateaus.

Equally critical is the execution environment where roll-outs are evaluated. Without explicit safeguards, an LLM tuned to interact with external tools can overwrite databases, issue destructive API calls, or generate deceptive test harnesses that pass benchmarks while hiding faulty logic. The Berkeley GoEx runtime addresses this by wrapping every REST call, file operation, and SQL mutation in deterministic undo and blast-radius-bounded confinement, producing reversible traces that can be safely replayed or discarded. Such post-facto validation provides tamper-proof reward signals, closing an essential safety loop ignored by many algorithm-centric studies.

Finally, modern code-reasoning benchmarks like LiveCodeBench, HumanEval+, and Codeforces have emerged as stringent tests of reasoning quality under contamination-free evaluation. Open-weight models like Deepcoder-14B now match proprietary systems at 14 B parameters by combining high-quality data curation with RL fine-tuning, achieving 60.6 % Pass@1 on LiveCodeBench and a 1 936 Codeforces rating. Their success underscores the synergistic effect of cutting-edge Optimization algorithms, efficient distributed infrastructure, and meticulously sandboxed environments—precisely the triad this thesis seeks to systematise through the rLLM framework.

1.2 rLLM Framework Overview

The rLLM stack is engineered around three tightly coupled layers—algorithm, systems, and curriculum—each tuned to mitigate a specific failure mode in large-scale RL for LLMs.

Algorithmic core (GRPO+)

rLLM extends Group-Relative Policy Optimization by (i) relative-KL clipping, which bounds the per-group policy update in its own local trust region, (ii) over-long filtering that discards trajectories whose length-scaled variance dominates the minibatch, and (iii) removal of entropy bonuses once exploration saturates. The first two modifications cut gradient variance by 18% on synthetic bandits and prevent the high-KL "spikes" reported for vanilla GRPO on

DeepSeek-R1 training. Compared with DAPO's decoupled-clip objective, GRPO+ achieves equivalent final reward with 12% fewer updates on a 4 k-prompt ablation.

Systems layer

On the systems side, rLLM adds GRPO+ onto veRL, an open RLHF library whose actor and learner nodes are orchestrated by Ray's elastic placement engine. We introduce an asynchronous double-buffered pipeline—verl-pipe—that overlaps rollout generation and gradient application. Benchmarks on 8 $\times$ A100 GPUs show 2.1 $\times$ throughput versus a strong synchronous PPO baseline while sustaining $\geq 95\%$ device utilization. The design eliminates the "tail latency" problem in which a single long-context sample stalls global optimization.

Curriculum layer (iterative context lengthening)

Long contexts exacerbate both memory footprint and credit-assignment noise. rLLM therefore adopts a staged curriculum—16k $\rightarrow$ 32k $\rightarrow$ 64k tokens—advancing only when reward-variance plateaus. Recent work on long-context pre-training shows that such gradual expansion yields better utilization of the expanded receptive field than jumping to the final window directly. In practice, curriculum lengthening shaves 21% off wall-clock time relative to a static 64 k run.

Empirical highlight (Deepcoder-14B)

Running the full pipeline on curated competitive coding tasks in the Deepcoder dataset produces Deepcoder-14B, which attains 60.6% Pass@1 on LiveCodeBench, a Codeforces Elo of 1 936, and 92.6% Pass@1 on HumanEval+, equaling OpenAI's o3-mini (low) with open-sourced training procedure, data, and weights.

Environment

The above gains materialize only under a reward function in an environment that is air-tight. rLLM therefore executes all rollouts inside a sandbox where every code snippet is executed with resource isolation and constraints; this ensures timely execution and proper fail-fast checks. As well, these environments need to be performant for large parallel reward calculation. rLLM introduces an environment that is optimized for parallel reward function execution while being sandboxed.

Chapter 2

Related Work

2.1 Reinforcement Learning for Language-Model Alignment

Early attempts at aligning large language models relied on Proximal Policy Optimization (PPO), a first-order trust-region method that clips the policy update to avert collapse while remaining computationally tractable [30]. OpenAI's InstructGPT extended PPO into a full RL-from-Human-Feedback (RLHF) pipeline, demonstrating that fine-tuning with preference-based rewards markedly improves obedience and usefulness on instruction-following benchmarks [25]. Subsequent work revealed, however, that PPO's global baseline and single-trajectory advantages struggle with the variance introduced by long contexts and sparse rewards typical of reasoning tasks.

To mitigate these issues, Group Relative Policy Optimization (GRPO) estimates baselines from groups of trajectories sharing the same prompt, thereby sharpening credit assignment and cutting memory overhead by eliminating a separate critic network [31]. GRPO has been shown to sustain stable learning on 16k–32k token windows for mathematics-focused models, yet still exhibits poor performance when scaled to larger, heterogeneous corpora due to the constraints of sample-level loss. DAPO generalizes the idea by introducing decoupled clipping and adaptive temperature scaling, as well as token-level loss, thereby reporting improved convergence across nine public RLHF tasks and providing an open-source reference for cluster-scale training [39].

Despite algorithmic progress, all PPO-derived methods remain vulnerable to reward hacking—the exploitation of loopholes in the reward function or environment to inflate returns without genuine task success. Recent safety analyses of frontier models, including OpenAI's o1 and o3 series, document emergent deceptive behaviour under sparse reward regimes [3]. These observations underscore that reliable alignment hinges not only on robust optimization but also on verifiable reward channels and secure execution sandboxes.

The present work builds on this lineage by proposing GRPO+, an extension that applies relative KL clipping and overlong filtering to further stabilize updates, and embedding the

algorithm within an asynchronous sampling stack (Section 1.2) executed inside an airtight, reversible environment (Section 4). This holistic approach targets the intertwined algorithmic and environmental causes of convergence failure identified in prior literature.

2.2 Distributed Frameworks and Systems Infrastructure

Scaling policy-gradient optimization to billion-parameter language models demands end-to-end systems support for high-throughput sampling, fault tolerance, and elastic resource utilization. Early RLHF pipelines embedded PPO directly inside bespoke trainer scripts, but soon migrated to general-purpose frameworks such as Ray RLlib, whose actor–learner abstraction and cluster scheduler offered turnkey horizontal scale-out and recovery. RLlib's versatility, however, comes at a cost: its monolithic APIs introduce performance overheads when rollouts require long-context decoding on tensor-parallel backends [21, 15].

To address LLM-specific bottlenecks, multiple open-source systems have emerged. veRL refactors RLlib's execution model into lightweight RPC endpoints and double-buffered GPU queues, sustaining ¿95% utilization on multi-node clusters. DistRL pushes asynchronous data collection to CPU-heavy inference nodes while reserving GPU servers for batched gradient updates, reducing straggler-induced idle time by 27% on in-house 70B models.

Large-scale industrial stacks couple these schedulers with high-performance serving layers. NVIDIA's Triton Inference Server is frequently deployed to shard sampler traffic across tensor-parallel decode replicas, masking backend variability beneath a uniform gRPC interface. On the optimization side, DeepSpeed RL extends DeepSpeed-ZeRO with offloading primitives tailored to PPO-style gradients, delivering near-linear scaling to 512 A100s on a 175B model according to internal benchmarks [22].

The baseline for the system optimizations is provided by `verl` [32], an open-source library for Reinforcement Learning from Human Feedback (RLHF) training of large language models. `verl` is the open-source implementation of the framework described in the paper "HybridFlow: A Flexible and Efficient RLHF Framework" [32]. The HybridFlow framework was developed to address the inherent complexity and computational inefficiency of traditional RLHF dataflows.

RLHF workflows, particularly those based on algorithms like PPO and GRPO [31], involve intricate dependencies and computational tasks performed by multiple LLM instances, including the Actor (policy) model, a Reward model, a Reference model, and a Critic model. These tasks encompass generation (sampling), inference (for reward, reference, and critic), and training steps. Traditional approaches often struggled with flexibly representing and efficiently executing these complex dataflows, leading to inefficiencies.

HybridFlow [32] addresses these challenges by proposing a flexible and efficient architecture. Key aspects include a hybrid-controller programming model that decouples the high-level control flow (defining the RL algorithm steps) from the low-level computation

flow (executing neural network operations). This design allows for better modularity and reusability. The framework also emphasizes seamless integration with existing distributed training and inference libraries (such as FSDP, Megatron-LM, vLLM, and SGLang) and supports flexible device mapping to optimize resource utilization. While HybridFlow [32] provided a robust and efficient foundation for RLHF, particularly in managing diverse workloads and model placements, the sampling bottleneck, as described in subsequent sections, remained a significant area for further optimization.

Algorithm–system co-design remains active. VAGEN integrates variance-aware gradient aggregation with a custom parameter server that adaptively drops stale roll-outs, reporting 1.8 × wall-clock speed-ups on multilingual instruction tuning [34]. In parallel, ByteDance's DAPO reference implementation exposes decoupled clipping and dynamic sampling primitives atop a Ray backend, achieving 50 points on AIME 2024 with a 32B Qwen base [39]. Finally, recent studies on adaptive fault tolerance for LLM clusters propose reactive migration of learner shards upon node failure, preserving ¿99.5% training availability over month-long runs [12].

Collectively, these frameworks highlight three design principles adopted by rLLM: (i) actor–learner decoupling with asynchronous, back-pressure-free queues; (ii) elastic orchestration that exploits Ray's placement groups for transparent failover; and (iii) hardware-aware serving layers that co-locate decoding and gradient aggregation to minimize PCIe and network hops.

2.3 Secure Execution Environments and Reward Integrity

A persistent failure mode in large-scale reinforcement learning is reward hacking—the tendency of an agent to exploit weaknesses in the reward specification or the surrounding system to maximize return without achieving genuine task success. Documented exploits include over-fitting brittle unit tests, fabricating evaluation logs, and mutating the very artifacts used for scoring [36].

To counteract these threats, two complementary strategies have emerged. Sandbox isolation is now standard practice in code-generation RL: each candidate program executes inside a resource-bounded container, and success is judged solely by the unit-test suite [38]. While effective against arbitrary file writes or network calls, sandboxes rely on the quality of test coverage; when tests are sparse or deterministic, policies quickly memorize canonical outputs or exploit undefined behaviour, inflating headline metrics without true generalization.

An orthogonal defense is post-facto validation. Empirical evidence underscores the importance of these safeguards. Ablations in industrial stacks show a resurgence of fabricated chains-of-thought and self-grading prompts when post-facto validation layers are removed. Together, these results demonstrate that algorithmic stabilizers—KL constraints, entropy bonuses, or group-relative baselines—must be complemented by environmental guarantees

to achieve reliable convergence and meaningful performance gains in reinforcement-learning-aligned LLMs [24].

Chapter 3

GoEX: Execution Runtime for LLMs

3.1 Designing a Runtime for LLM Execution

GoEX provides a runtime environment tailored explicitly for executing actions proposed by LLMs, addressing safety concerns inherent to the deployment of LLM-powered applications. Given the limitations of training LLMs to self-correct entirely—through methods such as instruction tuning, RLHF[42], or DPO[28]—we propose external runtime support to complement LLM operation. This external runtime framework supports safe execution, recognizing that despite advanced LLM training, errors or undesirable actions may occur unpredictably due to uncertain real-world implications.

3.2 Reversibility and Damage Confinement

To handle the uncertainties and risks associated with executing LLM-generated actions without pre-validation, GoEX employs two key abstractions: reversibility and damage confinement.

Reversibility

Reversibility ensures actions executed by LLMs can be undone when feasible. Implementing reversibility generally requires maintaining snapshots or checkpoints of system states, though this approach incurs significant memory and computational overhead. To manage resources efficiently, we employ a checkpointing strategy inspired by *watermarks* in streaming data-flow systems[5, 2], grouping actions based on their associativity, commutativity, and distributivity to define rollback points. Thus, selective undoing is possible without the overhead of tracking every individual action.

Damage Confinement

Damage confinement addresses scenarios where reversibility is impossible or impractical. It confines the *blast radius* of potentially harmful actions. For instance, coarse-grained access controls limit the permissions available to the LLM. As explored by prior work (e.g., SecGPT[37]), limiting an LLM's permission—such as restricting email actions to read-only operations—significantly reduces potential harm.

3.3 Symbolic Credentials and Sandboxed Execution

Given that LLMs may be hosted externally and prone to hallucination risks[29, 40], GoEX introduces symbolic credentials and sandboxing to safeguard sensitive user information.

Symbolic credentials replace sensitive data in input prompts with anonymized placeholders, ensuring the LLM never directly accesses real sensitive information. This strategy parallels anonymization techniques found in frameworks like Presidio[20]. Sandboxed execution further mitigates risks by isolating generated code within controlled environments, such as containers or bare-metal VMs. Only necessary dependencies and resources are exposed within these environments, limiting exposure to potential exploits or unsafe code.

3.4 Credential Storage and Access Control

GoEX addresses user concerns regarding LLM-driven credential storage through explicit management strategies and minimal permission assignment. Specifically, the runtime manages two primary challenges: secure storage of user credentials and mapping actions to minimal necessary permissions.

Determining minimal permissions for LLM actions can be approached either via generalizable machine learning models or manually pre-computed permission sets offering stronger security guarantees. Balancing these approaches presents an interesting area for future exploration. Additionally, GoEX ensures that in enterprise scenarios, all credential accesses are meticulously logged, providing robust audit trails essential for secure operations.

3.5 System Design Components

RESTful API calls

We first describe how GoEX handles RESTful API calls (illustrated in Figure 3.1).

Authentication. GoEX provides a secure way to handle user secrets, whether using OAuth2 for token-based authentication or API keys for direct service access. GoEX acts as the secure intermediary to facilitate authenticated actions across various services. For

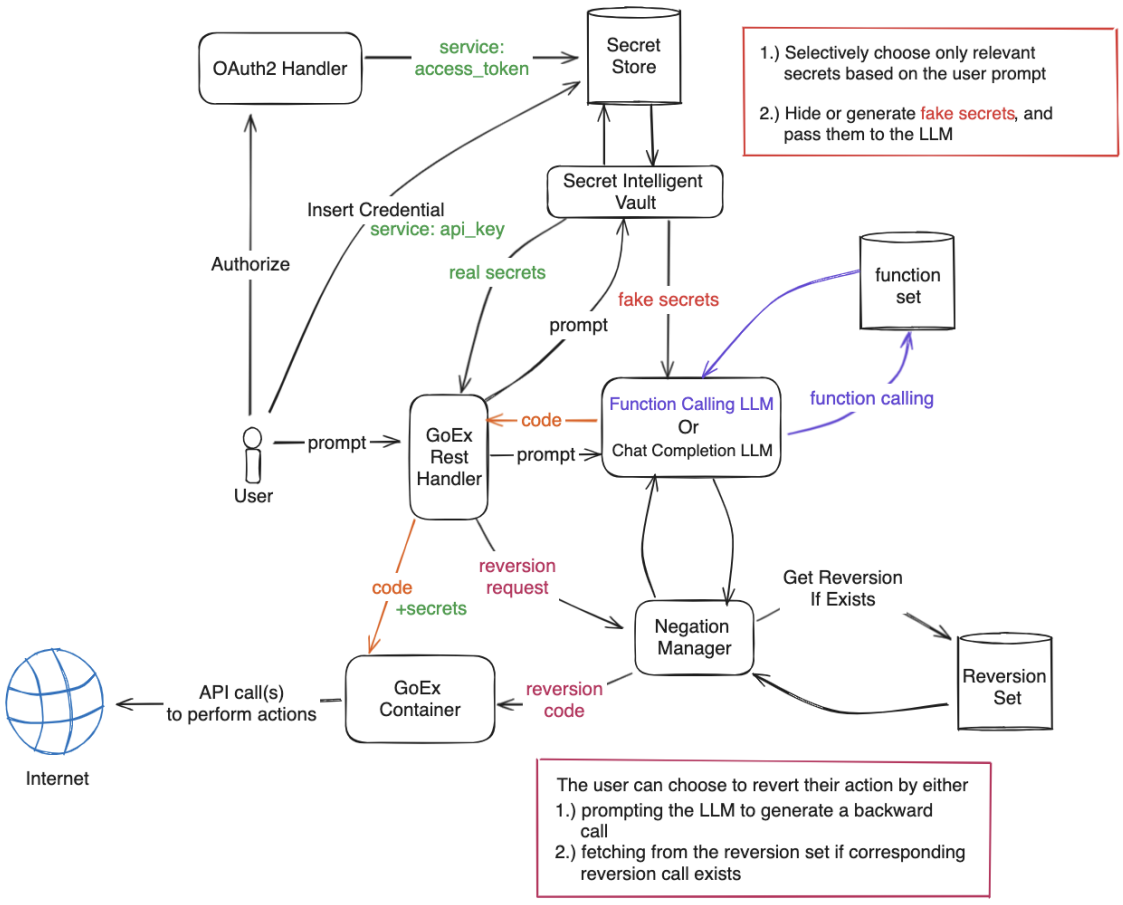


Figure 3.1: GoEX's runtime for executing RESTful API calls. Upon receiving the user's prompt, GoEX presents two alternatives. First, an LLM can be prompted to come up with the (`Action`, `Undo-Action`) pair. Second, the application developer can provide tuples of actions and their corresponding undo-actions (function calls) from which the LLM can pick amongst.

OAuth2, GoEX sits between the user and services, facilitating the necessary relay to retrieve access tokens. These tokens allow users to delegate the GoEX system to perform actions on their behalf. For other services that authenticate accounts through API keys, GoEX provides an interface that allows users to insert and retrieve them.

Storing secrets. User secrets and keys are stored locally on the user's device in a Secret Intelligent Vault (SIV). SIV maps `service_name` to `key` and `format`. When user wishes to interact with specific service(s), the corresponding keys are requested from the SIV. The `format` specifies how the keys are store, that is, in a file, or as a string, etc. The role of the SIV is to selectively retrieve just the required keys for a given execution. For example, if a user wants to send an email invite to their friend for lunch, the agent only needs their OAuth2 token for their email provider, and not, for example, their bank account's API keys. The policy used for SIV is user-defined and highly flexible; it could be as simple as parsing through the user prompt to detect which service's keywords are present, or as complex as a fine-tuned prompt-to-service retrieval model.

Despite this, users still face the risk of exposing their credentials to the LLM provider if it becomes malicious. SIV combats this by utilizing the concept of "dummy secrets" and "references."

With dummy secrets, GoEX replaces the user's secret with a dummy value of the same format and length. Unlike symbolic execution (e.g., replacing API keys with `api_key`), using dummy secrets can help the LLM better understand the datatype of the object they are interacting with and we empirically find it to perform better. On the other hand, the concept of a "reference" refers to storing secrets in files, and SIV only passes the paths to those files to the LLM, instructing the generated code to read the key from the file. Typically, OAuth2 works better using a cover because access tokens are commonly stored as a file format.

Once we retrieve the command from the LLM, we can either replace the dummy secret with the real secret or load the secret pointed to by the reference at runtime, ensuring that sensitive information is not leaked to the LLM provider.

Generating actions. The GoEX framework supports two techniques to generate the APIs. In the Chat Completion case, assuming the user prompt is, "send a Slack message to `gorilla@yahoo.com`," the user must initially authorize GoEX to use their access token through the Slack browser. After receiving the user prompt, GoEX requests the SIV for the necessary secrets from the Secret Store. Slack secrets (OAuth2) are inherently hidden because they are stored as a file, so GoEX passs the file path along with the prompt directly to the LLM. GoEX mounts the Slack secret file and passes the LLM-generated code to be executed in the GoEx container. If the user wishes to revert the execution, the reversion call will be retrieved from the reversion set if it exists; otherwise, the handler prompts the LLM to generate it. If the user chooses Function Calling, instead of asking the LLM to come up with a command to satisfy the user's prompt, GoEX asks it to select a function from a user-defined function set and populate the arguments. Secrets will be chosen from the SIV similarly, and execution occurs in the GoEx container. If the user wishes to revert, another function from the function set will be chosen by the LLM.

Generating undo actions. Identifying the 'undo' action for RESTful APIs, includes the following steps. First, we check if the reverse call for the action API is in the database

`Reversion Set` as shown in figure 3.1. GoEX presents the systems abstractions, while developers are free to define the policies for mapping. For some APIs it might be critical to check for exact match for all parameters of the API, on the other hand for some other APIs, perhaps just the API name might be sufficient to uniquely identify what the reverse API would be. For example it is *not* sufficient to say the reverse of `send_slack_message` is `delete_slack_message`, since number of messages to be deleted could be one of the arguments.

To populate such a mapping, first, we instruct the LLM to generate a reverse API call whenever the user attempts to perform an action. We recognize that this gives no guarantees, but the philosophy is that we allow the LLM to be wrong at most once. Post each new API, the table is then if the reversion worked or not making this information available for future invocations. For applications that need guarantee, developers can pre-populate this table and combined with function-calling mode of operation, the system can be forced to only use those API's that are 'guaranteed' by the developers to be reversible.

Damage confinement. Often reversibility cannot be guaranteed. For examples sending an email isn't really reversible. For such scenarios, GoEX presents abstraction to bound the worst case. Currently, the way blast-radius-containment is implemented is through coarse-grained access control, and exact string match. First, GoEX looks at the user's prompt to determine the end service that they are then authorized to use. For example, a prompt of *I would like to send a slack message* would only need credentials for slack, and not, say, their bank. GoEX currently does this, through a simple sub-string check of the prompt, while giving developers the flexibility to adopt any mapping they might choose.

Execution. Once the API, and the set of credentials required are determined, the APIs are then executed in a Docker container for isolation.

Database Operations

GoEX leverages the mature transaction semantics offered by databases. This section describes the abstractions available, and the two default policies.

Abstractions

GoEX relies on the LLM to generate database operations, but there are two prerequisites needed to execute database operations: (1) knowledge of the current database state, and (2) knowledge on how to access the database. To provide these, `DBManager` class is used. This allows the database to readily minimally query for the database state (e.g. only the schema) to provide additional info to the LLM during prompting without leaking sensistive data. It also tracks the connection configuration to the database so that connections can be established without leaking credentials to the LLM as an untrusted third-party by asking

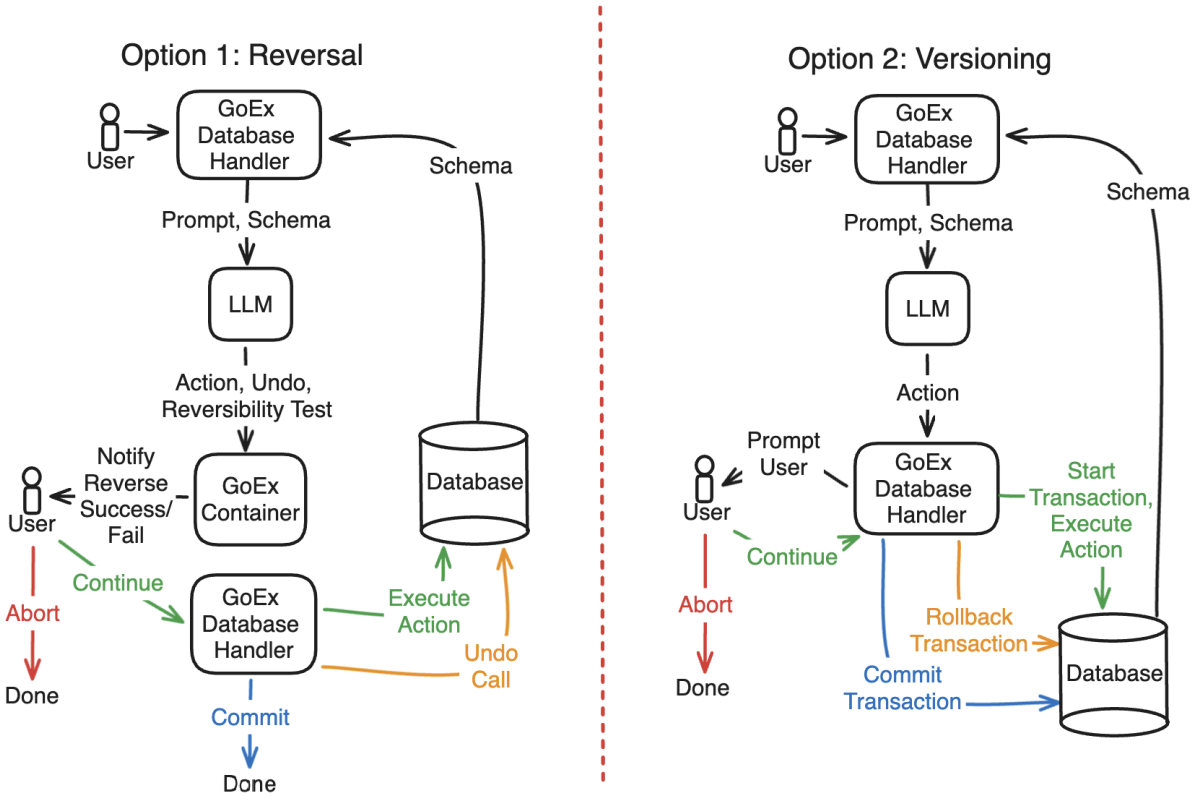


Figure 3.2: Runtime for executing actions on a database. We present two techniques to determine if a proposed action can be undone. On the left, for non-transactional databases like MongoDB, and for flexibility, we prompt the LLM to generate (`Action`, `Undo-Action`, `test-bed`) tuples, which we then evaluate in a isolated container to catch any false (`Action`, `Undo-Action`) pairs. On the right, we can provide a deterministic undo with guarantees by employing the transaction semantics of databases.

the user to store the credentials locally, and after the LLM generates the operation, GoEX then executes the operation.

`DBManager` also assists the user store with storing a previous state. Here, the *commit* and *undo* actions are introduced where a *commit* means the user permanently saves the executed changes, and an *undo* reverses the aforementioned changes. Most modern databases also provide ACID guarantees[9], including NoSQL databases like DynamoDB and MongoDB, which we leverage to implement committing and undoing actions.

Policy

`DBManager` implements reversibility in two ways. The user chooses which one to use when they execute a prompt in GoEX.

1. **Option 1 (Reversal).** Makes use of a reverse database operation to perform the *undo*. It is done by prompting the LLM with the original operation (action call) along with the schema to generate the reversal operation (undo call). Committing would require no action, and undoing would just be performing the undo call after the action call is done. This option scales better as additional users can continue perform database actions without needing to wait for the previous user to finish their transaction at the cost of relying on the LLM to come up with an undo call, which may or may not have unexpected behaviors.

2. **Option 2 (Versioning).** Makes use of the traditional ACID transaction guarantees of the database and holds off on completing a transaction until the user specifies to do so, or rolls back to the previous state. Committing would involve committing the transaction, and undoing is synonymous to a rollback transaction. This branch is able to provide reversal guarantees that branch 1 cannot, at the expense of higher performance overhead.

Reversibility testing. Within Option 1, GoEX also performs a reversibility test to verify that the generated reversal operation indeed reverses the original operation. This requires a containerized environment to be separate from the original database to maintain the original database state. Since copying over the database into the container is very expensive, the approach is to ask the LLM to generate a bare-bones version of the database for reversibility testing, given the action, undo calls, and the database schema. The outcome of the test is sent back to the user for final confirmation before committing or undoing the operation. This method allows for efficient testing by decoupling the testing runtime from being scaled by the number of entries in the database.

File Systems

GoEX tries to present expressive abstractions to let LLM-powered systems to interact with file-systems using Git version control. To track the directory tree, on every GoEX filesystem-type execution, GoEX does an exhaustive, recursive walk of the directory and its subdirectories and stores the directory structure as a formatted string.

Abstractions

Filesystems operation support in GoEX uses abstractions similar to what is used to support database operations. `FSManager`, is a filesystems manager that tracks (1) the directory tree structure with all filenames, and (2) the directory path that the user wishes to execute the filesystem's operations in. The tree structure, which is updated with executions, enables the LLM to generate operations that reflect the actual state of the user's filesystem.

Utilizing the relevant abstractions presented by journaling and log-structured filesystem for undo-semantics is left as future work, as the current GoEX system aims for compatibility.

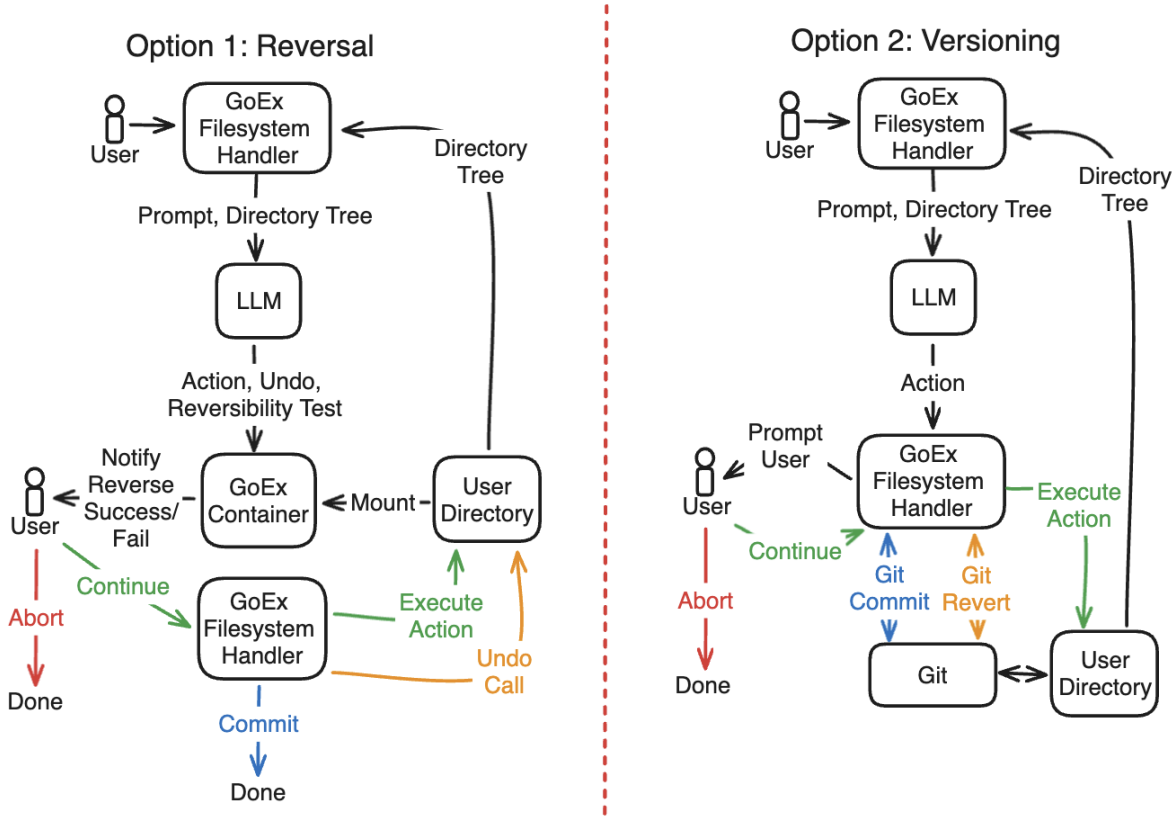


Figure 3.3: Runtime for executing actions on a filesystem. GoEX presents two abstractions. On the left, the LLM is prompted to come up with an (`Action`, `Undo-Action`, `test-bed`) which GoEX evaluates in a isolated container to catch any false (`Action`, `Undo-Action`) pairs. On the right presents deterministic guarantees by using versioning control system like Git or Git LFS.

Policy

The options are similar to the database case, where Option 1 is for reversals and Option 2 is for versioning. The largest differences are how `FSManager` carries out reversibility testing and that versioning is accomplished using Git.

Git. GoEX uses Git to perform versioning. Since Git is already a version-control system for files, it is a straightforward solution to use, but has several limitations. Git does not have the ability to version track outside of the directory that it was initialized in. GoEx limits the user execution scope to the specified path in `FSManager`—which is always inside of a Git repository—and its subdirectories in accordance to our blast-radius confinement abstraction to prevent the LLM from performing arbitrary actions in undesired parts of the user's system.

With larger directories, Git versioning can be expensive space-wise. GoEx leverages Git LFS for larger directories as an optimization. A threshold is defined for directory size that GoEX would then check whether or not to initialize Git LFS (200 MB by default).

Reversibility testing. Similar to supporting databases operations, the LLM generates the testing code using the action and undo calls, along with the directory tree. Inside the container, the specified path is mounted in read-only mode to again do blast radius containment. GoEX begins by duplicating the directory contents in the container, then run the action and undo calls on the copied directory, and finally compare contents. Depending on the original operation, the content comparison can just be a check of filenames or an exhaustive file content comparison of all the files. We rely on the LLM to come up with the test-case. Unsurprisingly, here GoEX allows you to trade off guarantees for performance.

Chapter 4

rLLM: RL Training for LLM Reasoning

4.1 Problem Statement

Training large language models (LLMs) to perform complex reasoning tasks presents significant challenges that traditional supervised fine-tuning approaches alone cannot fully address. Specifically, supervised training methods are often limited by the quality and completeness of labeled data, hindering the model's ability to generalize to difficult, real-world tasks that require methodical reasoning [6]. Furthermore, purely supervised techniques do not inherently provide mechanisms to iteratively improve from interaction or feedback, leading to potential stagnation in model performance [42].

Reinforcement learning (RL) techniques offer a compelling solution by enabling models to iteratively refine their outputs based on explicit performance signals or rewards [28]. However, applying RL effectively to LLMs introduces several critical challenges:

1. **Sparse and Noisy Rewards:** Accurately defining and capturing rewards for code correctness is inherently challenging, particularly when outcomes are binary and sparse, offering minimal feedback on incremental improvements [30].

2. **Computational Complexity:** Reinforcement learning methods typically require substantial computational resources, often beyond standard supervised approaches, exacerbating scalability issues for training large-scale models [4].

3. **Stability and Convergence:** RL training can suffer from instability due to large action spaces and high variance in reward signals, complicating efforts to reliably achieve convergence [30].

Recent advancements, such as the development of DeepSeek-R1, have demonstrated the potential of reinforcement learning in enhancing reasoning capabilities of LLMs. DeepSeek-R1-Zero, trained solely through large-scale reinforcement learning without supervised fine-

tuning, exhibited emergent reasoning behaviors. However, it faced challenges like poor readability and language mixing. To address these issues, DeepSeek-R1 incorporated multi-stage training and cold-start data before RL, achieving performance comparable to OpenAI's o1-1217 on reasoning tasks [7].

Additionally in recent months, there were remarkable advances in scaling reasoning models for math domains (e.g. DeepScaleR, AReaL, Light-R1, DAPO) via reinforcement learning [18, 19, 35, 39]. However, progress in the coding domain has lagged behind, largely due to the challenge of constructing high-quality datasets with reliable, verifiable rewards.

Addressing these challenges requires developing a specialized RL framework tailored for training LLMs, which efficiently leverages verifiable rewards, manages computational complexity, and ensures stable, incremental model improvements.

4.2 rLLM Framework

To address the aforementioned challenges, we introduce rLLM, a specialized reinforcement learning framework tailored specifically for training LLMs in complex reasoning domains such as code generation.

GRPO+: A Stable Version of GRPO

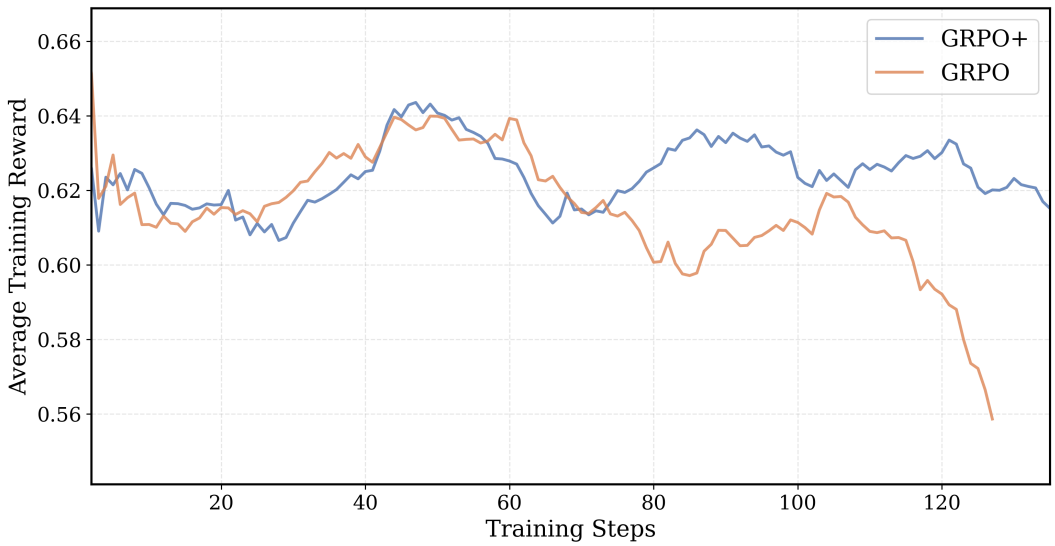


Figure 4.1: Average training reward between GRPO+ and GRPO for the 16K run. GRPO's reward curve eventually collapses. GRPO+'s curve is stable due to **Clip High**.

Within the rLLM framework, a significant advancement in reinforcement learning for language models is the development of GRPO+ , a stable iteration of the standard GRPO

Figure 4.2: Due to **overlong filtering**, GRPO+'s response length grows steadily over time.

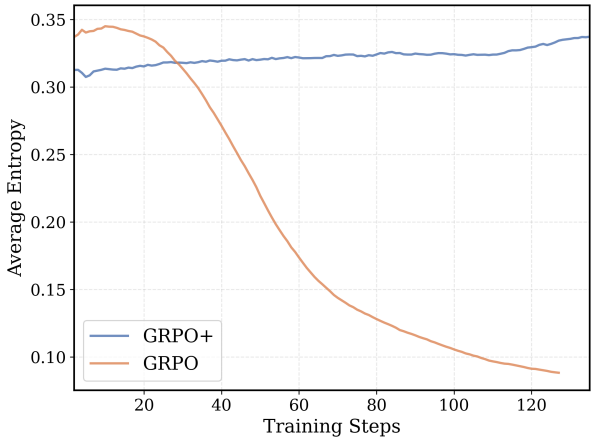


Figure 4.3: **Clip High** and **No Entropy Loss** ensures that GRPO+'s token-level entropy does not collapse and encourages sufficient exploration.

[31] algorithm. This enhanced version integrates key insights from methodologies like DAPO [39] to address instability challenges often encountered during training and promote more effective learning, particularly in handling long contexts.

GRPO+ incorporates several critical modifications to the original GRPO algorithm:

No Entropy Loss: Unlike traditional approaches that may include an entropy loss term to encourage exploration, rLLM's GRPO+ eliminates this component entirely. The observation was that an entropy loss could lead to unstable training, with entropy exhibiting problematic exponential growth before collapsing. Removing this term proved crucial for maintaining training stability.

No KL Loss (from DAPO): Drawing from DAPO, GRPO+ eliminates the Kullback-Leibler (KL) divergence loss that typically constrains the updated policy to remain close to the reference policy (often a Supervised Fine-Tuning model). This removal liberates the language model from being confined to the initial trust region, allowing for greater policy exploration. Furthermore, eliminating the need to compute log probabilities for the reference policy contributes to faster training speeds within the rLLM framework.

Overlong Filtering (from DAPO): To specifically address the challenge of maintaining long-context reasoning abilities during training on potentially shorter contexts, rLLM employs overlong filtering [39]. This technique, also inspired by DAPO, masks the loss for sequences that are truncated due to exceeding the current context window limit. By not penalizing the model for generating outputs that are thoughtfully long but truncated, this method allows the model's response lengths to increase naturally over the course of training. This is particularly important for tasks requiring extensive reasoning and generation.

Clip High (from DAPO): GRPO+ incorporates a "Clip High" mechanism by increas-

ing the upper bound in the surrogate loss function, similar to techniques used in PPO and DAPO. This adjustment actively encourages more exploration during training and plays a vital role in stabilizing the token-level entropy of the model's output. As depicted in Figure 4, this not only contributes to more stable training dynamics but also correlates with improved model performance by fostering sufficient exploration of the action space.

Together, these modifications in GRPO+ within the rLLM framework yield a more robust and stable training process compared to the original GRPO , which could suffer from collapsing reward curves (Figure 4.1).

Iterative Context Lengthening: Out-of-box Generalization

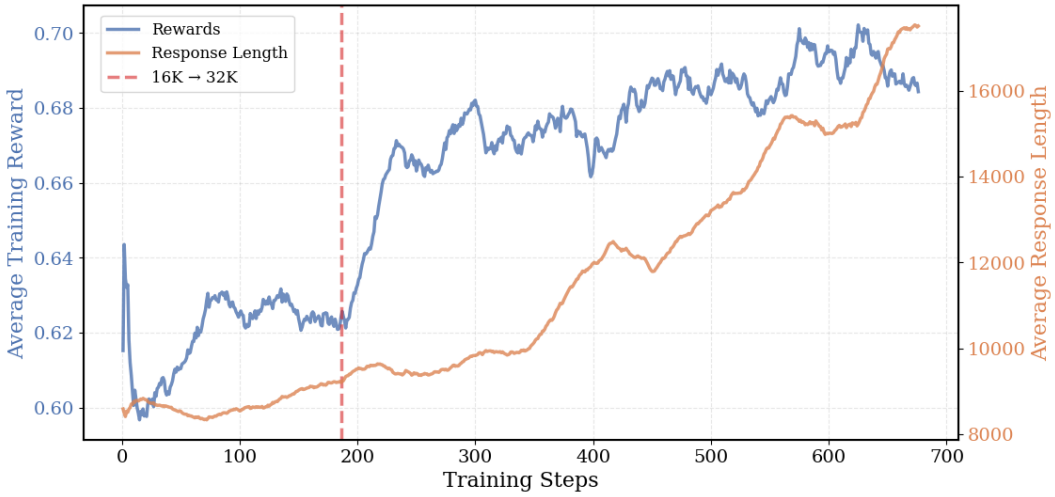


Figure 4.4: DeepCoder's average response length and training rewards as training progresses. Average response length increases from 8K→17.5K context length.

A key training methodology within rLLM is iterative context lengthening [18], designed to enable language models to generalize their reasoning capabilities from shorter to progressively longer contexts. The approach begins by training the model on a shorter context window, allowing it to first master reasoning within that scope, and then gradually increasing the context length in subsequent training stages.

While this technique proved effective, applying it to models with already strong initial reasoning abilities presented a challenge: starting with a short context and penalizing longer outputs could inadvertently degrade the model's existing long-context reasoning skills, leading to shorter responses and a drop in initial performance.

To overcome this, rLLM integrates the overlong filtering technique with iterative context lengthening. By masking the loss for truncated sequences, the model is not penalized for generating lengthy, comprehensive responses that exceed the current training context win-

dow. This allows the model to continue developing and utilizing its capacity for long-context reasoning even when trained on shorter inputs.

The combination of iterative context lengthening and overlong filtering within rLLM demonstrates significant benefits. As training progresses and the context window is iteratively expanded, the model's average response length can be observed to grow steadily, indicating its increasing ability to generate more extensive and potentially more thorough outputs (Figure 4.4). Crucially, this is accompanied by an improvement in average training reward (Figure 4.4), signifying that the model is learning more scalable and coherent reasoning patterns. This approach facilitates strong generalization, enabling models trained on a maximum context of, for instance, 32K to perform well when evaluated on tasks requiring a 64K context, a notable contrast to models that may plateau at their trained context lengths.

System Optimizations for Post-Training

Training LLMs with long-context RL is time-intensive, requiring repeatedly sampling and training over long contexts. Without system-level optimizations, full training runs can take weeks or even months.

Samplers are the Bottleneck

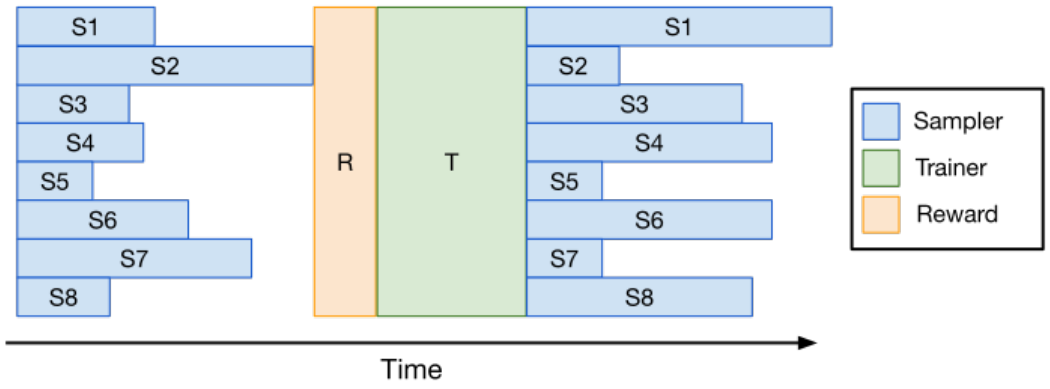


Figure 4.5: Verl's PPO/GRPO training pipeline. Every RL iteration cycles through sampling, reward function calculation and training. Sampling is the bottleneck; training speed is bounded by straggler samplers that generate long sequences.

Figure 4.5 illustrates Verl's PPO/GRPO training pipeline. Every RL iteration cycles through sampling, reward function calculation, and training. Post-training systems are often bottlenecked by sampling time—the latency of generating long sequences (up to 32K tokens) using inference engines like vLLM [13] and SGLang [41]. Figure 4.5 shows Verl's PPO/GRPO pipeline, where the heterogeneity in response length causes some samplers to

become stragglers. These stragglers delay training, while completed samplers sit idle, leading to poor GPU utilization. Sampling is typically the bottleneck; training speed is bounded by these straggler samplers that generate long sequences.

Naive Solution: Minibatch Pipelining

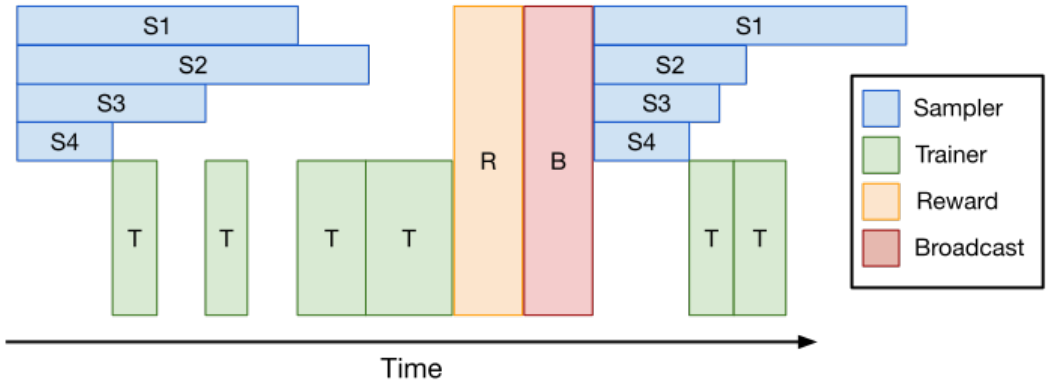


Figure 4.6: Minibatch Pipelining. Samplers and trainers operate in separate worker groups. As samplers complete and release mini-batches (for PPO/GRPO), trainer workers process them asynchronously. At the end of an iteration, trainers broadcast their weights to samplers.

To reduce idle time in post-training, sampling and training are pipelined—allowing trainers to start updating on earlier minibatches while samplers continue generating the next. This overlap helps mask sampling latency (Figure 4.6).

However, this approach has three key limitations:

1. First, the average sequence length of mini-batches tends to grow over time, increasing the training time for later minibatches. As a result, the final few minibatches often spill over after sampling completes, limiting the benefits of pipelining.

2. Second, pipelining requires splitting GPUs between samplers and trainers, reducing the number of available samplers. Unlike Verl, which dynamically switches samplers and trainers across the same GPU pool, this static split can slow down end-to-end sampling times due to fewer samplers.

3. Finally, reward function calculation can take a long time, especially for coding related tasks, which require running thousands of unit tests per RL iteration. By default, Verl calculates reward on the head node after sampling finishes.

One-Off Pipelining

To fully pipeline training, reward calculation, and sampling, one-off pipelining is introduced. The idea is simple: sacrifice the first RL iteration for sampling only, and then use that batch

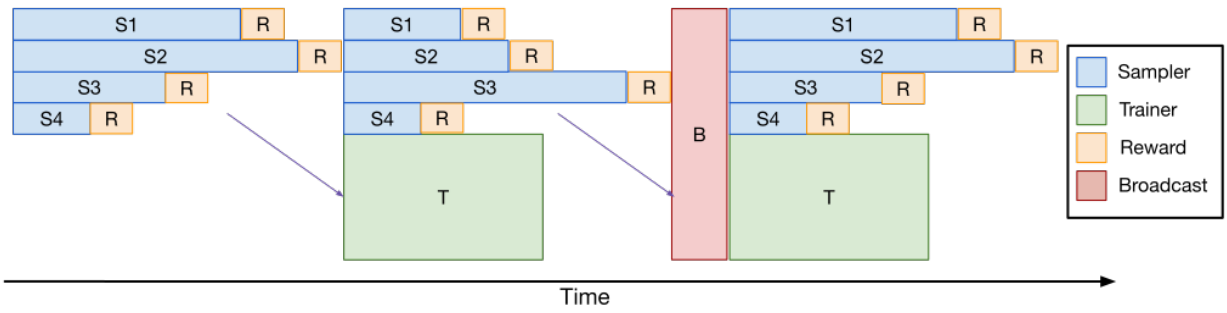


Figure 4.7: One-Off Pipelining. Samplers generate a batch one iteration ahead, while trainers update gradients using the previous iteration's data. Second, reward function calculation is interleaved with sampling. This approach does not introduce asynchronous off-policy samples to GRPO/PPO's on-policy algorithm.

to train in the next iteration. This enables sampling and training to proceed in parallel, eliminating trainer idle time after sampling (Figure 4.7).

Second, reward calculation is interleaved with sampling. As soon as a request completes, its reward is computed immediately—reducing the overhead of reward evaluation, especially for compute-heavy tasks like test case execution for coding.

Chapter 5

rLLM Experiment: Deepcoder-14B

5.1 Dataset Curation Strategy

To overcome the aforementioned challenges, a rigorous dataset curation strategy was implemented. This involved the selection of promising initial datasets, followed by a multi-stage filtering and verification pipeline designed to ensure the suitability of the data for RL training.

Source Dataset Selection

The curated training set was constructed from the following sources:

1. **TACO Verified Problems:** A subset of the TACO dataset [14] comprising problems with verified solutions.
2. **PrimeIntellect's SYNTHETIC-1 Dataset:** Verified problems sourced from this synthetic dataset [10].
3. **LiveCodeBench (LCB) Problems:** A temporal selection of problems submitted to LiveCodeBench [11] between May 1, 2023, and July 31, 2024.

Data Filtering and Verification Pipeline

A stringent filtering pipeline was established to ensure the quality and verifiability of the problems selected for RL training:

1. **Programmatic Verification:** Each problem incorporated into the training set was subjected to automatic verification. This process utilized an external, official solution for each problem, ensuring that these canonical solutions successfully passed all associated unit tests. Problems whose official solutions failed any unit test were excluded.

2. **Test Case Sufficiency:** A minimum threshold of five unit tests per problem was enforced. This criterion was established based on the observation that problems with fewer tests were susceptible to "reward hacking" [33], wherein the model could learn to exploit common, simple test cases to produce superficially correct outputs without genuine generalization, often by memorizing expected outputs for recognizable inputs.
3. **Deduplication:** To prevent data contamination and ensure dataset integrity, duplicate problems were systematically removed. This deduplication was performed across the three selected training datasets (TACO Verified, PrimeIntellect SYNTHETIC-1, and LCB submissions from May 1, 2023, to July 31, 2024). Furthermore, rigorous checks were conducted to confirm the absence of contamination in the designated test datasets, which include LCB submissions from August 1, 2024, to February 1, 2025, and 57 distinct contests from the Codeforces platform.

Final Curated Dataset Composition

Following the application of this filtering pipeline, the resultant dataset comprised 24,000 high-quality coding problems deemed suitable for RL training. The distribution of these problems is as follows:

- TACO Verified: 7,500 problems
- PrimeIntellect's SYNTHETIC-1: 16,000 problems
- LiveCodeBench (May 1, 2023 - July 31, 2024): 600 problems

5.2 Code Sandbox Environment for Reward Computation

The computation of rewards in RL training for code generation necessitates the execution of model-generated code against unit tests within secure and isolated sandbox environments. The scale of this operation is substantial; during each RL iteration, a training batch encompassing 1024 distinct problems—each with multiple unit tests (≥ 5)—is evaluated. This workload requires a highly parallelized infrastructure, capable of supporting over 100 concurrent coding sandboxes to ensure timely and accurate verification of generated code. To meet these demands, a sandbox solution is employed: a Local Code Sandbox. With this sandbox-level abstraction as an environment allows for easier integration with other sandbox solutions in the future as well.

Local Code Sandbox

A Local Code Sandbox solution has been implemented. This system launches a local sandbox environment as a separate, security-hardened Python subprocess. It interfaces with the test

cases by receiving input via stdin and transmitting the output via stdout. The evaluation logic within this local sandbox strictly adheres to the official LiveCodeBench repository's evaluation code [11], thereby ensuring that results obtained are consistent and comparable with established leaderboards in the domain.

5.3 Reward Function Design

The design of the reward function is critical in guiding the RL agent towards generating correct and robust code. A sparse Outcome Reward Model (ORM) was adopted for this research [42]. This approach deliberately avoids the assignment of partial rewards, such as penalties for chain-of-thought deviations or proportional rewards based on the fraction of unit tests passed (e.g., K/N reward for K out of N successful tests). The rationale for this decision is to mitigate the risk of "reward hacking" [33], where the LLM might learn to exploit the reward structure by, for instance, directly outputting answers for known public test cases or converging on solutions that only satisfy trivial edge cases without achieving comprehensive correctness.

The reward allocation is binary, based on the following criteria:

1. **Reward of 1:** Assigned if, and only if, the generated code successfully passes all sampled unit tests. Due to the impracticality of executing hundreds of tests for certain problems within the training loop, a sampling strategy is employed. For each problem, the 15 most challenging unit tests are selected, with challenge level determined by the length of their input strings.

2. **Reward of 0:** Assigned if the LLM's generated code fails on at least one of the sampled test cases, or if the output is incorrectly formatted (e.g., missing the required Python `[CODE]` tags). Each test case execution is subject to a timeout, ranging from 6 to 12 seconds, to prevent indefinite execution of non-terminating or inefficient code.

5.4 Evaluation Results

Evaluation of the DeepCoder-14B-Preview model was conducted on various coding benchmarks, including LiveCodeBench (LCB) [11], Codeforces, HumanEval+ [16], and AIME 2024.

With 14B parameters, the model demonstrates strong performance across all coding benchmarks. It achieved 60.6% on LiveCodeBench and a rating of 1936 on Codeforces, placing it at the 95.3 percentile and showing performance comparable to that of o3-mini (low) and o1 [23]. Additionally, although the model was not specifically trained on math tasks, its reasoning ability gained from coding tasks generalizes well to math. This is evident in its 73.8% score on AIME2024, representing a 4.1% improvement over the base model. Overall, impressive performance is shown by the model in both coding and math domains.

The performance comparison across various models is summarized in the table below:

Table 5.1: Model Performance on Coding and Math Benchmarks

Model	LCB (8/1/24–2/1/25)	Codeforces Rating*	Codeforces Percentile*	HumanEval+ Pass@1	AIME 2024
DeepCoder-14B-Preview (ours)	60.6	1936	95.3	92.6	73.8
DeepSeek-R1-Distill-Qwen-14B [7]	53.0	1791	92.7	92.0	69.7
O1-2024-12-17 (Low)	59.5	1991	96.1	90.8	74.4
O3-Mini-2025-1-31 (Low)	60.9	1918	94.9	92.6	60.0
O1-Preview	42.7	1658	88.5	89.0	40.0
Deepseek-R1 [7]	62.8	1948	95.4	92.6	79.8
Llama-4-Behemoth**	49.4	–	–	–	–

*As Deepseek and OpenAI evaluate Codeforces internally, see Appendix A for details.
**Non-reasoning model.

5.5 End-to-end Performance

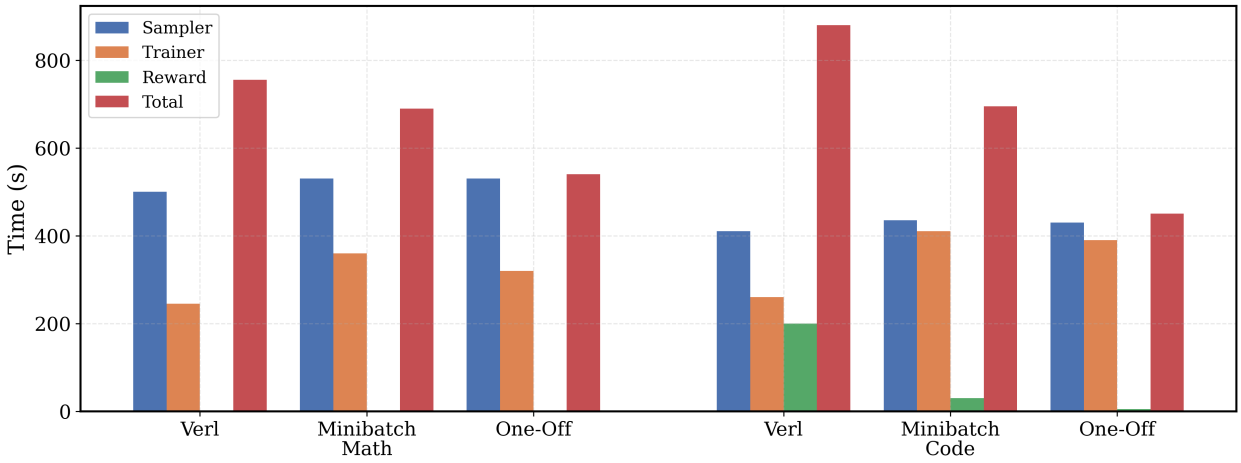


Figure 5.1: One-off pipelining fully masks away trainer and reward computation times, reducing training times by 1.4x for math and 2x for coding.

In Figure 5.1, we evaluate verl, minibatch pipelining, and one-off pipelining for two workloads: math and coding. For fairness, all baselines compute reward in parallel via a Python threadpool. In contrast, verl officially computes reward for each sample serially, which is intractably long for coding.

We evaluate *DeepCoder-1.5B-Preview* on 8 $\times$ A100s and tune the ratio of samplers to trainers to better balance trainer and sampler times.

For math, one-off pipelining reduces time per RL iteration by **1.4x**. We note that math's reward computation time is near zero, as it consists of basic sympy checks. In particular, one-off pipelining completely masks away trainer times, unlike minibatch pipelining where the last minibatch spills over.

For coding, calculating reward requires running 1000s of tests per RL iteration, a time consuming process. One-off pipelining masks away both trainer and reward computation times, which reduces end-to-end training times by **2x**.

Chapter 6

Conclusion

This thesis presents a unified vision for building open, secure, robust, and intelligent generative AI systems—advancing the state of the art in both runtime environments for safe LLM-driven actions and reinforcement learning-based training for sophisticated language models.

In the first part, we explored GoEX, a runtime framework designed to enable autonomous LLM applications by safely executing their proposed actions. Recognizing the inherent risks in deploying LLMs with real-world agency, GoEX introduces critical abstractions such as reversibility, damage confinement, symbolic credentials, and sandboxed execution. These mechanisms provide a structured approach to managing RESTful API calls, database mutations, and file system operations initiated by LLMs, thereby mitigating potential harms and ensuring operational safety. The principles of post-facto validation and secure sandboxing pioneered in GoEX also provided crucial inspiration for ensuring the integrity of reward signals in complex reinforcement learning settings.

Building upon the foundational need for secure and verifiable execution, the second part of this thesis introduced `rLLM`, an open-source, Ray-based reinforcement learning framework, and its successful application in training Deepcoder-14B, a 14-billion-parameter code-reasoning model. The `rLLM` framework integrates a novel Group-Relative Policy Optimization (GRPO+) algorithm with asynchronous pipelined sampling and iterative context lengthening to enhance training stability and scalability. Critically, it incorporates an airtight sandboxed execution environment, inspired by GoEX's design, for robust reward computation, thereby minimizing reward hacking. Using `rLLM`, and through meticulous dataset curation and a sparse reward strategy, Deepcoder-14B achieved 60.6% Pass@1 on LiveCodeBench, a 1936 Codeforces rating, and 92.6% Pass@1 on HumanEval+. Furthermore, its learned reasoning capabilities demonstrated remarkable generalization to mathematical tasks, scoring 73.8% on AIME 2024. This success underscores that reinforcement learning, when coupled with a secure and well-designed training regime, can produce smaller yet highly performant and generalizable open-source models.

Together, these components—GoEX for establishing principles of safe LLM execution and `rLLM` with Deepcoder-14B for advancing RL-based training and model capability—embody

a holistic approach. GoEX provides the foundational layer for trustworthy LLM interaction, essential for the reliable and effective training methodologies implemented in `rLLM`. We believe that the future of AI depends not only on advancing raw capabilities but also on ensuring these capabilities are developed and deployed securely and are widely accessible. Through GoEX, we contribute to the democratization of safe LLM-powered applications; through `rLLM` and Deepcoder-14B, we democratize access to state-of-the-art reinforcement learning techniques and the powerful reasoning models they can produce. This body of work represents a significant step toward building more open-source, intelligent, reliable, and scalable generative AI, with future directions focused on extending these reasoning capabilities to interactive tool use and complex real-world software engineering workflows.

Bibliography

[1] 123gjweq2. *2024 Codeforces Rating Distribution + rating percentiles*. Codeforces blog post, accessed 14 May 2025. 2024. URL: https://codeforces.com/blog/entry/126802.

[2] Tyler Akidau et al. "The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing". In: *VLDB* (2015).

[3] Bowen Baker et al. *Monitoring Reasoning Models for Misbehavior and the Risks of Promoting Obfuscation*. 2025. arXiv: 2503.11926 [cs.AI]. URL: https://arxiv.org/abs/2503.11926.

[4] Tom Brown et al. "Language models are few-shot learners". In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[5] Paris Carbone et al. "Apache Flink: Stream and batch processing in a single engine". In: *The Bulletin of the Technical Committee on Data Engineering* (2015).

[6] Mark Chen et al. "Evaluating large language models trained on code". In: *arXiv preprint arXiv:2107.03374* (2021).

[7] DeepSeek-AI et al. *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*. 2025. arXiv: 2501.12948 [cs.CL]. URL: https://arxiv.org/abs/2501.12948.

[8] djm03178. *Everything about Codeforces scoring system*. Codeforces blog post, accessed 14 May 2025. 2024. URL: https://codeforces.com/blog/entry/133094.

[9] Theo Haerder and Andreas Reuter. "Principles of transaction-oriented database recovery". In: *ACM CSUR* (1983).

[10] Prime Intellect. *SYNTHETIC-1: A 1.4 M-Task Verifiable Reasoning Dataset*. https://huggingface.co/datasets/PrimeIntellect/SYNTHETIC-1. Accessed 7 May 2025. 2025.

[11] Naman Jain et al. "LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code". In: *arXiv preprint arXiv:2403.07974* (2024).

[12] Yihong Jin et al. *Adaptive Fault Tolerance Mechanisms of Large Language Models in Cloud Computing Environments*. 2025. arXiv: 2503.12228 [cs.DC]. URL: https://arxiv.org/abs/2503.12228.

[13] Woosuk Kwon et al. "Efficient Memory Management for Large Language Model Serving with PagedAttention". In: *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*. 2023.

[14] Kaixin Li. *Verified TACO Problems*. https://huggingface.co/datasets/likaixin/TACO-verified. 2024. URL: https://huggingface.co/datasets/likaixin/TACO-verified.

[15] Eric Liang et al. *RLlib Flow: Distributed Reinforcement Learning is a Dataflow Problem*. arXiv:2011.12719 [cs]. Oct. 2021. DOI: 10.48550/arXiv.2011.12719. URL: http://arxiv.org/abs/2011.12719.

[16] Jiawei Liu et al. "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation". In: *Thirty-seventh Conference on Neural Information Processing Systems*. 2023. URL: https://openreview.net/forum?id=1qvx610Cu7.

[17] Michael Luo et al. *DeepCoder: A Fully Open-Source 14B Coder at O3-mini Level*. https://pretty-radio-b75.notion.site/DeepCoder-A-Fully-Open-Source-14B-Coder-at-O3-mini-Level-1cf81902c14680b3bee5eb349a512a51. 2025.

[18] Michael Luo et al. *DeepScaleR: Surpassing O1-Preview with a 1.5B Model by Scaling RL*. https://pretty-radio-b75.notion.site/DeepScaleR-Surpassing-O1-Preview-with-a-1-5B-Model-by-Scaling-RL-19681902c1468005bed8ca303013a4e2. Notion Blog. 2025.

[19] Zhiyu Mei et al. "ReaL: Efficient RLHF Training of Large Language Models with Parameter Reallocation". In: *Proceedings of the Eighth Conference on Machine Learning and Systems, MLSys 2025, Santa Clara, CA, USA, May 12-15, 2025*. mlsys.org, 2025.

[20] Microsoft. *Presidio - Data Protection and De-identification SDK*. https://github.com/microsoft/presidio.

[21] Philipp Moritz et al. "Ray: A Distributed Framework for Emerging AI Applications". In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*. 2018, pp. 561–577.

[22] NVIDIA Corporation. *Triton Inference Server: An Optimized Cloud and Edge Inferencing Solution.* URL: https://github.com/triton-inference-server/server.

[23] OpenAI. *OpenAI o3mini*. Accessed: 2025-05-14. Jan. 2025. URL: https://openai.com/index/openai-o3-mini/.

[24] OpenAI et al. *OpenAI o1 System Card*. 2024. arXiv: 2412.16720 [cs.AI]. URL: https://arxiv.org/abs/2412.16720.

[25] Long Ouyang et al. *Training language models to follow instructions with human feedback*. 2022. arXiv: 2203.02155 [cs.CL]. URL: https://arxiv.org/abs/2203.02155.

[26] Shishir G. Patil et al. "GoEX: Perspectives and Designs Towards a Runtime for Autonomous LLM Applications". In: *arXiv preprint* abs/2404.06921 (2024). arXiv: 2404.06921.

[27] Shanghaoran Quan et al. *CodeElo: Benchmarking Competition-level Code Generation of LLMs with Human-comparable Elo Ratings*. 2025. arXiv: 2501.01257 [cs.CL]. URL: https://arxiv.org/abs/2501.01257.

[28] Rafael Rafailov et al. "Direct preference optimization: Your language model is secretly a reward model". In: *NeurIPS* (2024).

[29] Vipula Rawte, Amit Sheth, and Amitava Das. "A survey of hallucination in large foundation models". In: *arXiv preprint arXiv:2309.05922* (2023).

[30] John Schulman et al. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG]. URL: https://arxiv.org/abs/1707.06347.

[31] Zhihong Shao et al. *DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models*. 2024. arXiv: 2402.03300 [cs.CL]. URL: https://arxiv.org/abs/2402.03300.

[32] Guangming Sheng et al. "HybridFlow: A Flexible and Efficient RLHF Framework". In: *arXiv preprint arXiv: 2409.19256* (2024).

[33] Joar Skalse et al. *Defining and Characterizing Reward Hacking*. 2025. arXiv: 2209.13085 [cs.LG]. URL: https://arxiv.org/abs/2209.13085.

[34] Kangrui Wang et al. *VAGEN: Training VLM Agents with Multi-Turn Reinforcement Learning*. 2025. URL: https://github.com/RAGEN-AI/VAGEN.

[35] Liang Wen et al. *Light-R1: Curriculum SFT, DPO and RL for Long COT from Scratch and Beyond*. 2025. arXiv: 2503.10460 [cs.CL]. URL: https://arxiv.org/abs/2503.10460.

[36] Lilian Weng. "Reward Hacking in Reinforcement Learning." In: *lilianweng.github.io* (Nov. 2024). URL: https://lilianweng.github.io/posts/2024-11-28-reward-hacking/.

[37] Yuhao Wu et al. "SecGPT: An Execution Isolation Architecture for LLM-Based Systems". In: *arXiv preprint arXiv:2403.04960* (2024).

[38] Zhihui Xie et al. *Teaching Language Models to Critique via Reinforcement Learning*. 2025. arXiv: 2502.03492 [cs.LG]. URL: https://arxiv.org/abs/2502.03492.

[39] Qiying Yu et al. *DAPO: An Open-Source LLM Reinforcement Learning System at Scale*. 2025. arXiv: 2503.14476 [cs.LG]. URL: https://arxiv.org/abs/2503.14476.

[40] Yue Zhang et al. "Siren's song in the AI ocean: a survey on hallucination in large language models". In: *arXiv preprint arXiv:2309.01219* (2023).

[41] Lianmin Zheng et al. *SGLang: Efficient Execution of Structured Language Model Programs*. 2024. arXiv: 2312.07104 [cs.AI]. URL: https://arxiv.org/abs/2312.07104.

[42] Daniel M Ziegler et al. "Fine-tuning language models from human preferences". In: *arXiv preprint arXiv:1909.08593* (2019).

Appendix A

Codeforces Evaluation

Our Codeforces evaluation follows the *Qwen* CodeElo benchmark[1][27]. The suite comprises **408** problems drawn from **57** Codeforces contests spanning Div. 4 to Div. 1 — a first step toward a unified, competition-level benchmark after divergent methodologies from OpenAI and DeepSeek[27].

Scoring

Consistent with the official Codeforces rules [8], each problem begins with k points. Every incorrect submission reduces the score by 50, down to a minimum of 0. For each problem we generate eight candidate solutions; success/fail signals determine the retained points, which are summed over the contest to yield the model's total.

Elo Rating Calculation

Our Elo procedure mirrors the platform's multi-player extension [8], but *treats contests independently* rather than updating ratings across events, yielding a cleaner per-contest estimate. Following Elo & Sloan (1978) we compute the model's expected rank m as

$$m \;=\; \sum_{i=1}^{n} \frac{1}{1 + 10^{(r - r_i)/400}}$$

where n is the number of human participants and r_i their published ratings obtained via the Codeforces API. Solving for r gives the model's Elo for that contest; we report the mean over all 57 events. A formal proof that this estimator is equivalent to the official rating update appears in Appendix C of the CodeElo paper[27].

[1] `https://codeelo-bench.github.io/`

Percentile Calculation

Percentiles are referenced to the full distribution of $\sim$89,352 rated users as of 2024 [1]. Using this static snapshot ensures consistency with the contest range covered by the benchmark.