Degree Project in Machine Learning

Second Cycle, 30 credits

Learning Agency in the Terminal with Repository-Level Reinforcement Learning

BJARNI HAUKUR BJARNASON

Authors

Bjarni Haukur Bjarnason <bhbj@kth.se>
Electrical Engineering and Computer Science
KTH Royal Institute of Technology

Place for Project

Stockholm, Sweden
KTH

Examiner

Martin Monperrus <monperrus@kth.se>
Division of Theoretical Computer Science
KTH Royal Institute of Technology

Supervisor

André Afonso Nunes Silva <andreans@kth.se>

Division of Theoretical Computer Science

KTH Royal Institute of Technology

Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
APR	Automated Program Repair
DAPO	Decoupled Clip and Dynamic Sampling Policy Optimization
GPU	Graphics Processing Unit
GRPO	Group Relative Policy Optimization
GSPO	Group Sequence Policy Optimization
JSON	JavaScript Object Notation
KL	Kullback-Leibler
LLM	Large Language Model
LoRA	Low-Rank Adaptation
MLP	Multi-Layer Perceptron
MoE	Mixture of Experts
NAISS	National Academic Infrastructure for Supercomputing in Sweden
NCCL	NVIDIA Collective Communications Library
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
SFT	Supervised Fine-Tuning
VRAM	Video Random Access Memory

Abstract

Software engineering agents have shown strong real-world debugging capabilities, yet a core mismatch persists between multi-step, interactive deployment and training that uses static datasets of isolated code changes. This thesis presents a fully open, end-to-end system for online, execution-free Reinforcement Learning (RL) that trains Large Language Models (LLMs) inside a coding agent scaffold (Nano). The system uses live weight synchronization via NCCL to push policy updates to running inference servers during training. While industry systems exhibit agentic coding capabilities consistent with agent-based training, their methods remain undisclosed. This work provides an open, reproducible recipe.

We train using Group Sequence Policy Optimization (GSPO) with a light Kullback-Leibler (KL) regularizer on a 1,000-task curriculum spanning ten programming languages, completing in 144 Graphics Processing Unit (GPU)-hours on 3 A100s. On SWE-Bench-Verified, patch submission rates (non-empty patches) rise from 37% to 78% and mean patch-similarity rewards increase by 54%, while test-verified success remains approximately flat at 6–7%. These results establish that online, execution-free RL reliably improves agent operational competence within academic compute budgets. Translating these gains to functional correctness likely requires longer training or alternative reward design. We release all infrastructure, methodology, and evaluation protocols to enable reproducible study of online RL for interactive coding agents.

Keywords

automated program repair, coding agents, reinforcement learning, group relative policy optimization, group sequence policy optimization, online learning, tool-augmented language models, execution-free training, multilingual generalization

Sammanfattning

Kodagenter har visat starka verkliga felsökningsförmågor, men en grundläggande diskrepans kvarstår mellan flerstegs, interaktiv implementation och träning som använder statiska dataset av isolerade kodändringar. Denna avhandling presenterar ett helt öppet, end-to-end-system för online, exekveringsfri RL som tränar LLMs inuti en kodagentskall (Nano). Systemet använder live viktsynkronisering via NCCL för att skicka policyuppdateringar till körande inferensservrar under träning. Medan industrisystem uppvisar agentiska kodningsförmågor som överensstämmer med agentbaserad träning, förblir deras metoder hemliga. Detta arbete tillhandahåller ett öppet, reproducerbart recept.

Vi tränar med GSPO med en lätt KL-regularisator på ett 1,000-uppgifters läroplan som spänner över tio programmeringsspråk, och slutför på 144 GPU-timmar på 3 A100s. På SWE-Bench-Verified ökar patch-inlämningsfrekvenser (icke-tomma patchar) från 37% till 78% och genomsnittliga patch-likhetsbelöningar ökar med 54%, medan testverifierad framgång förblir ungefär platt vid 6–7%. Dessa resultat fastställer att online, exekveringsfri RL pålitligt förbättrar agentoperativ kompetens inom akademiska beräkningsbudgetar. Att översätta dessa vinster till funktionell korrekthet kräver sannolikt längre träning eller alternativ belöningsdesign. Vi släpper all infrastruktur, metodik och utvärderingsprotokoll för att möjliggöra reproducerbar studie av online RL för interaktiva kodagenter.

Nyckelord

automatisk programreparation, kodagenter, förstärkningsinlärning, grupprelativ policyoptimering, gruppsekvens-policyoptimering, online-inlärning, verktygsförstärkta språkmodeller, exekveringsfri träning, flerspråkig generalisering

Acknowledgements

I would like to express my sincere gratitude to my supervisor, André Afonso Nunes Silva, and my examiner, Martin Monperrus, for their exceptional guidance throughout this thesis. Their supervision extended beyond technical feedback: they trusted me to pursue an ambitious research direction and supported a complete methodological pivot midway through the project when evidence suggested a more promising path.

This research would not have been possible without access to the computational infrastructure provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS). Training on the Alvis and Berzelius GPU clusters enabled experimentation at a scale far beyond typical academic resources, making the investigation of online RL for coding agents feasible. I am grateful to the infrastructure providers and the Swedish research community for maintaining these shared resources.

I thank the members of ASSERT-KTH who regularly attended the Monday meetings, which operated as an informal book club for those interested in Artificial Intelligence (AI) for software engineering. These sessions provided a welcoming forum to discuss recent papers, share emerging ideas, and engage with a group of like-minded researchers.

I acknowledge the developers and maintainers of the open-source projects used in this research: TRL and the broader HuggingFace ecosystem, vLLM, DeepSpeed, PyTorch and more.

Lastly, I thank my friends at KTH, Jonas and Navin, who have been with me from the start of my master's studies. Their steady friendship and camaraderie are the foundation on which these studies stand.

Contents

Chapter 1

Introduction

Automated program repair has rapidly transformed from a nascent research area to one of the most promising applications of agentic AI systems. By "agentic," we refer to systems that can autonomously interact with environments through tool use, make sequential decisions based on observations, and iteratively refine their approach based on feedback, as opposed to single-pass generation models that produce outputs without environmental interaction. Recent advances in autonomous software engineering have achieved remarkable performance milestones, with systems like Claude 4 now exceeding 70% success rates on SWE-Bench-Verified [1]. This rapid progress underscores the critical importance of maintaining open research capable of understanding frontier capabilities. A broad consensus has emerged among researchers: competitive access to training methodologies is essential to ensure that critical insights remain within the scientific community rather than concentrated in proprietary systems. Recent open weight releases such as Kimi-K2 [26] and Qwen3-Coder [27] follow closely behind frontier performance, but do not include full training recipes. This reinforces the need for open, reproducible methods and analysis.

This thesis directly addresses this need by investigating how a single, minimalist agent called Nano enables a model to acquire agentic behavior through experiential RL feedback occurring entirely within the agent's interactive loop. By situating all learning and adaptation within the Nano agent, we provide clear, reproducible insights into the mechanisms by which RL cultivates autonomy and effective tool use in repository-level code repair.

An important contemporaneous effort is SWE-RL [29], which applies Group Relative Policy Optimization (GRPO) with patch-similarity rewards (using Python's `difflib.SequenceMatcher`) in a static, single-turn setting where full file contexts are provided, the model outputs a patch directly, and similarity to ground truth defines the reward signal. Our work is complementary. We also use execution-free patch-similarity rewards but combine them with GSPO, a more robust sequence-level variant, in an agentic, multi-turn regime where the model must first acquire relevant context through tool use before proposing edits. This distinction between static one-shot patch generation and interactive context acquisition motivates our focus on agent-integrated training within a single minimalist scaffold rather than prompt-time scaffold engineering.

The necessity and transformative potential of agentic training emerge from examining the historical limitations of supervised learning for code repair and the foundational role of LLMs in enabling RL for complex tasks.

1.1 Background

Traditional systems for automated code repair have relied predominantly on supervised learning from static datasets of bug-fix pairs. In this paradigm, models passively observe input-output mappings between broken and corrected code, treating bug fixing as a pattern recognition task rather than engaging with the full interactive software engineering process. While computationally efficient, this approach fundamentally misrepresents the nature of repository-level software debugging, which involves iterative hypothesis refinement, strategic exploration of entire codebases, execution of diagnostic commands, and contextual interpretation of error signals across multiple rounds of investigation. Moreover, the information needed to fix a bug in real-world repositories is rarely contained within the bug-fix pair itself, necessitating the interactive capabilities that coding agents provide to navigate and understand the broader codebase context.

The question becomes: how do we train such agentic systems when the very complexity that necessitates their existence also makes traditional training approaches intractable? RL has achieved remarkable results in well-structured domains like Go, where clear success metrics exist and every action represents a valid move within the game's rules [24]. However, RL fundamentally requires that learning systems can meaningfully participate in the task from the beginning. They must be capable of generating valid actions, interpreting feedback, and understanding the basic structure of the problem space. For complex reasoning tasks like software debugging, this prerequisite participation threshold has historically been insurmountable.

LLMs have altered this fundamental constraint by providing the world knowledge and reasoning capabilities necessary to attempt tasks of unprecedented complexity. When equipped with appropriate interfaces, these models can navigate file systems, interpret error messages, execute diagnostic commands, and generate syntactically valid code modifications. This capability foundation transforms previously intractable problems into domains where RL becomes feasible. In effect, LLMs provide the first rung on the ladder that makes applying RL to software engineering feasible.

However, unlike traditional RL agents that directly produce discrete actions from observations (e.g., chess moves, joystick inputs), LLMs fundamentally operate as text generators. They cannot directly perceive environments or execute actions; they require an intermediary translation layer that converts generated text into executable actions and translates environmental responses into textual observations.

This architectural distinction necessitates a different approach to RL. In this paradigm, a lightweight agent scaffold interprets tool calls from the LLM, executes corresponding actions in the environment, and returns textual observations for processing. The entire system combining the LLM and scaffold forms the RL agent, with learning occurring through this mediated interaction cycle. This thesis instantiates exactly one such scaffold (Nano agent) and conducts all training strictly within this single-agent setting.

When equipped with appropriate scaffolds, LLMs can navigate file systems, execute terminal commands, interpret error messages, and generate syntactically correct code modifications. This capability transformation makes it practical to apply RL to real-world software engineering tasks, as models can now generate the rich interaction trajectories required for policy gradient estimation. Terminal-oriented agency benchmarks such as TauBench [34] provide standardized evaluation frameworks for such capabilities.

Training adopts an execution-free approach where rewards derive from patch similarity rather than test execution, enabling language-agnostic training without maintaining test infrastructure for each programming language. Concrete observation, action, and reward definitions are specified in Chapter 3.

The significance of this paradigm shift extends beyond technical feasibility. Human software developers do not repair bugs through single-shot pattern matching; they engage in exploratory processes involving hypothesis formation, incremental information gathering, and iterative refinement. They navigate complex codebases, examine interconnected components, execute diagnostic commands, and progressively construct mental models of system behavior. This active learning process differs fundamentally from passive pattern recognition, suggesting that training models through interactive experience may yield qualitatively superior debugging capabilities.

This direction aligns with widely observed advances at the frontier: agentic capabilities are clearly improving, with public reports of stronger coding performance and enhanced interactive capabilities. These developments heighten the need for open, reproducible understanding of how agentic training works, rather than reliance on closed systems. Our work addresses this need by studying RL with interactive coding agents in an open setting.

1.2 Problem Statement

The central problem is twofold: understanding how online RL for multi-turn tool-using agents is done and making it feasible within academic compute constraints. Frontier labs have likely addressed these challenges. Systems like Anthropic's Claude Code and OpenAI's Codex demonstrate capabilities suggesting specialized agent training, but their methods remain proprietary. The investment in purpose-built agent scaffolds indicates successful training approaches exist, yet without disclosed methodologies, the scientific community cannot replicate these results or understand how they were achieved.

Even when methodologies are partially disclosed, computational and operational demands place such training beyond most academic budgets, as pipelines involve dozens of interacting components built atop complex LLM foundations. This concentration of both knowledge and resources in proprietary labs creates fundamental asymmetries that undermine the open research ecosystem: without access to tractable training recipes, the scientific community cannot critically evaluate these approaches, replicate findings, or build upon established methods. As online RL for interactive agents emerges as a promising direction, bridging both the knowledge gap and the resource gap becomes essential for broad research participation.

We therefore develop a low-resource, execution-free RL recipe for repository-level Automated Program Repair (APR). The investigation addresses three research questions:

RQ1: Does execution-free GSPO training converge across diverse programming languages?

- Primary indicators: overall reward progression and variance evolution across the 1,000-task curriculum (750 Python + 250 multilingual tasks); per-language reward improvements from epoch 1 to epoch 2 on identical problem instances across nine programming languages.
- Analysis: reward mean and standard deviation trends for convergence validation; epoch-over-epoch comparison per language to assess language-agnostic learning.
- Hypothesis: execution-free patch-similarity rewards provide sufficient signal for policy convergence despite not directly optimizing for functional correctness, and this convergence generalizes across diverse programming languages without language-specific engineering.

RQ2: How does GSPO training improve Nano agent scaffold adaptation?

- Primary indicators: tool-call success rate, action efficiency, and command usage evolution over training.
- Analysis: pre-training vs. post-training comparison of scaffold-level efficiency metrics and qualitative examination of changes in command usage patterns.
- Hypothesis: RL training improves scaffold interaction efficiency and tool-use reliability, with observable shifts in command patterns toward more effective tool use.

RQ3: Does execution-free patch-similarity RL training improve SWE-Bench-Verified performance?

- Primary metric: test-verified success rate on SWE-Bench-Verified (approximately 500 Python debugging tasks).

- Comparison: pre-training baseline vs. post-training performance using identical evaluation protocol.
- Hypothesis: training on static patch-similarity rewards improves functional bug-fixing success despite not directly optimizing for test passage.

1.3 Contributions

This research makes several methodological and technical contributions to enable RL training for coding agents.

We employ GSPO [36], a sequence-level policy optimization algorithm that extends the group-relative baseline principle with improved importance weighting for variable-length sequences. GSPO demonstrates superior robustness compared to earlier group-relative methods (GRPO, DAPO), proving particularly effective for small effective batch sizes and high-variance rewards characteristic of multi-turn agent training under academic compute constraints.

Training proceeds through iterative online interaction: the agent explores repositories, generates repair episodes, and receives rewards based on static patch-similarity comparison between its repository modifications and ground-truth patches.

Primary comparisons report pre-training baseline vs. post-training performance on SWE-Bench-Verified using the same Nano agent scaffold for both evaluations, eliminating confounds from supervised fine-tuning or prompt engineering. Measured outcomes include repair success rates, reward progression, tool-call validity, action efficiency, and command usage patterns.

Data collection captures complete agent-environment interactions: tool invocations, repository states, reward signals computed via patch-similarity comparison, training dynamics, and aggregated evaluation metrics. Training uses a 1,000-task curriculum (750 SWE-Gym Python tasks + 250 SWE-Bench-Multilingual tasks), with primary evaluation on SWE-Bench-Verified and multilingual generalization assessed through per-language reward progression across training epochs.

This methodology enables quantitative validation and mechanistic insights into how RL shapes interactive debugging behavior through the Nano agent.

1.4 Delimitations

This investigation operates within carefully defined boundaries that balance scientific rigor with practical constraints. Understanding these delimitations is essential for interpreting our findings and their applicability to broader contexts.

Task Scope: This work addresses automated bug repair where known errors exist and the objective is patch generation. Evaluation focuses on the Nano agent operating within its native scaffold, while broader code-generation benchmarks (HumanEval, MBPP) remain out of scope. This focused investigation enables deep analysis of interactive debugging behavior while acknowledging that specification-based program synthesis constitutes orthogonal research.

Language and Environment Constraints: Training uses a 1,000-task curriculum (750 SWE-Gym Python issues + 250 SWE-Bench-Multilingual issues) curated to take advantage of our programming language agnostic approach.

Compute Environment Constraints: Experiments execute on Berzelius and Alvis Swedish national GPU clusters using SLURM scheduling without Docker-based container orchestration support. This infrastructure constraint shaped both Nano agent design and training pipeline architecture, favoring SLURM-compatible parallelization and process-level isolation over container-based approaches.

Model Architecture Scope: Our primary development and analysis focuses on Qwen3-14B—a hybrid reasoning model with strong tool-calling capabilities—with reasoning explicitly disabled to conserve context window budget in multi-turn episodes. Limited comparisons with Qwen3-8B, Qwen3-32B, and Llama3.1-8B appear in the Appendix to illustrate capacity and architecture effects, but full model ablation studies are left for future work.

Evaluation Protocol Boundaries: Rewards derive from automated patch-similarity comparison rather than test suite execution, trading functional verification for computational tractability and reproducibility. This approach aligns with established benchmarks and enables large-scale multilingual training, though it may undervalue functionally equivalent patches with syntactic divergence from ground truth. Evaluation emphasizes single-commit bug repairs; multi-stage refactoring and architectural changes remain out of scope.

Agent Implementation: The Nano agent adopts a deliberately minimalist design, exposing only essential shell commands and file patching operations (two tools: `shell`, `apply_patch`). This simplicity enables clean experimental isolation of RL effects, though richer tool sets might exhibit different performance-complexity trade-offs.

Methodological Boundaries: We briefly explored distillation from proprietary models via Supervised Fine-Tuning (SFT) but abandoned this direction due to limited empirical gains and philosophical misalignment with open research objectives focused on training from environmental interaction rather than imitating closed systems.

1.5 Outline

The remainder of this thesis progresses from theoretical foundations through empirical validation to conclusions:

Chapter 2: Background and Related Work reviews policy optimization algorithms for language models (Proximal Policy Optimization (PPO), GRPO, DAPO, GSPO), situates APR within repository-level evaluation paradigms, and contrasts agentless vs. agentic approaches to motivate our execution-free, agent-integrated training methodology.

Chapter 3: Method formalizes the Nano agent (observation/action spaces, interaction loop, termination), specifies training data (SWE-Gym, SWE-Bench-Multilingual), defines the patch-similarity reward formulation, describes GSPO policy optimization with masked loss, and outlines the evaluation protocol addressing RQ1-RQ3.

Chapter 4: Infrastructure documents the practical realization: vLLM-based serving with tool calling, live adapter synchronization via NCCL, trainer extensions for multi-turn GSPO, SLURM orchestration, compute-efficient infrastructure (Low-Rank Adaptation (LoRA), DeepSpeed ZeRO, gradient checkpointing, BF16), and training protocol execution.

Chapter 5: Results presents experimental findings: training convergence across languages with policy gradient signal stability (RQ1), scaffold adaptation metrics and behavioral changes (RQ2), and SWE-Bench-Verified performance (RQ3). The chapter concludes with a discussion interpreting the observed patterns and their implications.

Chapter 6: Conclusions summarizes contributions, discusses design philosophy and lessons learned (system complexity, reward misspecification, infrastructure challenges), examines broader implications for agentic AI, identifies limitations, and proposes future research directions for RL-trained coding agents.

Chapter 2

Theoretical Background and Related Work

This chapter establishes the theoretical and empirical foundations for applying online RL to APR with tool-augmented LLMs. We begin by situating APR historically, then introduce a taxonomy of LLM-based approaches—scaffold-free, agentless, and agentic—that clarifies which paradigms support end-to-end policy learning. With the APR landscape established, we review the instruction-driven datasets that enable training and evaluation of repository-level repair systems. We then provide an in-depth treatment of policy optimization algorithms, progressing from PPO through group-relative methods to GSPO, followed by parameter-efficient adaptation via LoRA, establishing the theoretical toolkit for the methodology in subsequent chapters.

2.1 Automated Program Repair before LLMs

Empirical APR research prior to LLMs evaluated primarily on test-suite-based Java benchmarks and function-level algorithmic benchmarks. *Defects4J* established reproducible evaluation with full-project test suites, while *GitBug-Java* expanded repository-level benchmarking with additional real-world projects and build configurations [7, 23]. Despite these repository-level resources, repair methods of that era commonly focused on local search spaces and single-file patches, reflecting the computational constraints and prevailing algorithmic approaches of the time. *QuixBugs* operates at function granularity: standalone Java/Python programs with a single seeded defect per program and accompanying unit tests [11].

Three families of approaches dominated this era. *Generate-and-validate* systems search mutation spaces, accepting candidates that satisfy test suites; GenProg pioneered this approach, while PAR demonstrated pattern-guided edit selection [8, 10]. *Semantics-based* methods employ program analysis (symbolic execution, constraint solving) to synthesize expressions at instrumented program locations, exemplified by SemFix [15]. *Learning-guided* approaches incorporate statistical priors to prioritize search; Prophet learns patch ranking models from historical fixes [13]. These methods established feasibility on test-suite benchmarks while revealing limitations of test adequacy as a correctness proxy: overfitting to specific test cases and brittle generalization to untested scenarios. These constraints, combined with difficulty reasoning beyond localized contexts, motivated the repository-grounded, model-based approaches that emerged with LLMs.

2.2 Automated Program Repair with LLMs

Codex marked an inflection point: large-scale pretraining on code enabled practical repair and synthesis, shifting APR from symbolic search to learned edit distributions [3]. Frontier laboratories subsequently converged on repository-grounded evaluation (SWE-Bench, SWE-Bench-Verified) and deployed first-party agent scaffolds (OpenAI Codex CLI, Anthropic Claude Code, Google Gemini CLI) that orchestrate tool invocations over live repositories [2, 4, 6, 16, 17].

We organize LLM-based repair by inference-time autonomy: (i) *scaffold-free* models that generate patches without tool execution, (ii) *agentless* systems where external scripts curate context and apply edits, and (iii) *agentic* systems where models autonomously plan, invoke tools, and iterate based on observations.

2.2.1 Scaffold-Free

Scaffold-free approaches frame repair as translation from buggy code to fixed code. The model receives a localized and self-contained context, typically a snippet or a single file with a narrow edit window, and proposes a patch in one interaction turn. There is no repository traversal, no tool use during generation, and no execution signal in the loop. Any tests are run outside the model after the fact.

Within this regime we focus on post-training methods that adapt an existing LLM for repair. A representative example is *RepairLLaMA*, which fine-tunes a general code model on curated bug-fix pairs so that it proposes minimal and targeted edits without relying on an execution scaffold [22]. A parallel thread brings RL into the same scaffold-free setting at the function or snippet level. *CodeRL* optimizes code generation with unit test execution as feedback while operating on localized contexts [9].

These setups are attractive because data and pipelines are simple, adaptation is straightforward, and inference is fast. It provides an effective baseline for measuring translation quality in localized contexts. Its limitations follow from the design: without an execution loop the model cannot test hypotheses, observe runtime behavior, or coordinate edits across files. Repository state and build systems remain external to generation.

2.2.2 Agentless

Moving beyond scaffold-free translation, *agentless* systems typically operate over a live repository while keeping execution outside the model. A thin scaffold retrieves relevant files, builds diffs, and structures the prompt; the model returns patch blocks or edit intents; a human or scripted scaffold runs tests and shell commands and applies the edits. This preserves repository grounding and repeatability without autonomous tool use.

We treat agentless systems as interfaces rather than agents: they make the contributions of context curation and patch formatting explicit and easy to ablate. In practice, components inside the scaffold (retrievers, rankers, patch validators) can be trained, and logs from these pipelines support offline learning or imitation without exposing the model to a full tool loop. Representative analyses show that a simple three-stage pipeline (localize, repair, validate) can match or exceed more complex setups when the scaffold is well-engineered [30]. Related work begins to introduce reinforcement signals in repository settings while still short of a fully autonomous terminal loop, e.g., SWE-RL for repository-level reasoning [29].

The limitation is structural. Because execution remains outside the model, the scaffold decides what to surface, when to run tests, and how to apply or revert edits. The model does not learn the decision-making that drives repair. What to try next, when to backtrack, when to submit or abandon a patch; because those choices live in the interface rather than in the policy. The reward signal therefore entangles scaffold choices with model output, and the resulting histories are not suitable for end-to-end training. Agentless setups are excellent for reproducible evaluation and for collecting logs, but they are a poor fit when the goal is to train a model to use tools by itself.

Moreover, many agentless systems invoke LLMs as stateless functions rather than maintaining continuous conversation histories. Between invocations, the system modifies or restarts context windows to inject new information, apply filters, or enforce structural constraints—interventions that break the causal chain necessary for gradient-based learning. When conversation histories are frequently pruned, reordered, or regenerated by the outer scaffold, the model cannot learn from the natural consequences of its prior outputs. This architectural choice optimizes for immediate task performance through careful prompt engineering but renders end-to-end policy learning infeasible, as the training signal would conflate the model's decisions with the scaffold's context manipulation.

2.2.3 Agentic

Agentic systems operate directly over a live repository and let the model initiate actions. The model can open and edit files, run tests and shell commands, read raw outputs, and iterate until a patch is ready. Public research scaffolds make this concrete: *SWE-Agent* formalizes an agent-computer loop for software tasks, *OpenHands* provides a general terminal-based platform for coding agents, and *mini-swe-agent* shows that a small, disciplined tool set in a tight loop is effective [28, 31, 32].

Operationally, these systems rely on structured tool calls, typically via JSON. The model emits a well-formed function call with arguments, an external executor runs it, and the resulting output is returned to the model.

```
{
  "function_name": "apply_patch",
  "arguments": {
    "file_path": "src/utils.py",
    "old_content": "def buggy_function():",
    "new_content": "def fixed_function():"
  }
}
```

Figure 2.2.1: Example JSON tool call structure for an apply_patch function.

This approach transforms LLMs from passive text generators into systems capable of interacting with their environment by invoking tools and observing their results.

This shift has also shaped how models are packaged and evaluated: recent open-weight releases pair model checkpoints with first-party scaffolds and report agent-style evaluations, while disclosing only partial training details. We summarize representative examples.

Qwen3–Coder (Alibaba/Qwen team). Qwen3–Coder is positioned as an *agentic* coding model and ships with a first-party command-line scaffold (*Qwen Code*) that drives repository-grounded autonomous workflows [19, 27]. The release materials emphasize agent behavior in realistic development loops and demonstrate how the paired scaffold surfaces those capabilities. At the same time, the post-training recipe, data generation, reward design, and optimization details is described only at a high level. Weights and the scaffold are available, but a reproducible end-to-end training pipeline is not.

Kimi–K2 (Moonshot AI). Kimi-K2 details an *agentic* data synthesis pipeline paired with a joint reinforcement learning stage that mixes real and synthetic environment interactions. [26] The stated goal is to shape multi-step decision making—tool selection, iterative trial-and-error, and revision—rather than single-pass text generation. The report frames this as necessary for repository-grounded coding and other multi-step tasks. As with Qwen3, model artifacts and high-level methodology are public, but *no* training recipe.

GLM–4.5 (Zhipu AI). GLM–4.5 positions itself as an *agentic, reasoning, and coding* (ARC) family and explicitly reports a comprehensive post-training phase that combines expert iteration with SFT and RL, aimed at strengthening multi-step agentic tool use. [25]. The public release provides open-weight checkpoints but does *not* include a reproducible end-to-end agentic RL training pipeline or environment traces.

Taken together, these open-weight efforts converge on the same direction: agent-facing post-training that couples tool use, long-context coding, and multi-turn control. They document SFT and RL training stages for agentic post-training and evaluations, but stop short of releasing reproducible training stacks. In contrast, recent work such as *DeepSWE* publicly reports an RL-trained coding agent within a terminal environment and releases open-weight models alongside practical training details, offering a rare end-to-end reference point for open agentic RL on software tasks. [14]

In practical terms, agentic systems close the loop between hypothesis and verification: the model requests an action, observes the real outcome, and can adjust. This makes online RL feasible and aligns the learning signal with the deployed behavior. By contrast with agentless setups, the model is exposed to the full decision process of repair rather than a curated slice, so the resulting histories are suitable for end-to-end training.

2.3 Training and Evaluation Datasets

Instruction-driven datasets provide natural language issue descriptions alongside buggy repository states and ground-truth patches, enabling both training and rigorous evaluation of repository-level repair systems. Each instance specifies the debugging task through a problem description, enabling agents to learn from how developers communicate bugs rather than from test failure signals alone.

2.3.1 Training Datasets

SWE-Gym [18] provides approximately 2,400 Python bug-fixing tasks extracted from real GitHub repositories, serving as a large-scale training resource for repository-level repair. SWE-Bench-Multilingual [33] extends instruction-driven debugging beyond Python, offering tasks spanning nine programming languages: Rust, Java, PHP, Ruby, JavaScript, TypeScript, Go, C, and C++.

2.3.2 Evaluation Benchmark

SWE-Bench-Verified [17] serves as the community evaluation standard, containing approximately 500 carefully validated Python debugging tasks with deterministic test outcomes. The benchmark measures functional correctness through actual test execution, providing ground-truth signals for whether proposed fixes resolve reported issues without introducing regressions.

2.4 Reinforcement Learning for Language Models

This section reviews policy optimization algorithms for LLMs, progressing from PPO to group-relative policy optimization (GRPO), and concluding with GSPO, which we ultimately adopt for its superior robustness in multi-turn agent training. We first establish PPO notation, then present the group-relative baseline principle introduced by GRPO along with subsequent refinements (Dr. GRPO, DAPO), and conclude with GSPO's sequence-level importance weighting that addresses stability challenges in variable-length trajectories.

2.4.1 Preliminaries

An autoregressive language model with parameters θ is treated as a policy π_θ over sequences. Let $\mathcal{D}$ denote a set of queries, with $x \in \mathcal{D}$, and let $y = (y_1, \dots, y_{|y|})$ be a tokenized response. The sequence likelihood factorizes as

$$\pi_\theta(y \mid x) \;=\; \prod_{t=1}^{|y|} \pi_\theta\big(y_t \mid x,\, y_{<t}\big). \tag{2.1}$$

A reward function r assigns a scalar outcome to a completed pair, with $r(x, y) \in [0, 1]$. Expectations are taken over $x \sim \mathcal{D}$ and over responses sampled from a frozen behavior policy $\pi_{\theta_{\text{old}}}$ that generated the data for the current update. The operator clip(w) denotes clipping importance ratio w into the symmetric range $(1 - \epsilon,\ 1 + \epsilon)$ controlled by $\epsilon > 0$; when asymmetric clipping is used, we write the bounds explicitly. A frozen reference model is denoted π_{ref} and a KL penalty to this reference is written $\beta\, D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}})$ where $\beta \geq 0$ controls the penalty strength.

Each method's original design with respect to KL regularization is preserved in the presentation: PPO, GRPO, and Dr. GRPO include an optional KL penalty to π_{ref}, whereas DAPO and GSPO do not.

2.4.2 Proximal Policy Optimization

PPO performs multiple gradient steps on samples drawn from the old policy while constraining update size through clipping [20]. With per-token importance ratio

$$w_t(\theta) \;=\; \frac{\pi_\theta(y_t \mid x, y_{<t})}{\pi_{\theta_{\text{old}}}(y_t \mid x, y_{<t})}, \tag{2.2}$$

and averaging across the sequence length, the objective is

$$J_{\text{PPO}}(\theta) = \mathbb{E}\left[\frac{1}{|y|}\sum_{t=1}^{|y|}\min\big(w_t\,\hat{A}_t,\ \text{clip}(w_t)\,\hat{A}_t\big)\right] - \beta\,D_{\text{KL}}(\pi_\theta\|\pi_{\text{ref}}). \tag{2.3}$$

The averaging by $|y|$ normalizes for length so long responses do not dominate the minibatch objective. We use terminal outcome supervision with no discounting, so the return at every prefix equals the final outcome $R := r(x, y) \in [0, 1]$:

$$G_t = R \quad \text{for all } t \in \{1, \dots, |y|\}, \qquad \hat{A}_t = G_t - V_\phi(x, y_{<t}) = R - V_\phi(x, y_{<t}). \tag{2.4}$$

Intuitively, the advantage contrasts the realized outcome with what the value model expected at each prefix: positive values increase the probability of y_t, negative values decrease it, and values near zero leave it largely unchanged.

Maintaining a separate value model comparable in size to the policy substantially increases the memory footprint during training; this overhead is exacerbated for long responses, as both models must process extended sequences in parallel.

2.4.3 Group Relative Policy Optimization (GRPO)

GRPO replaces the learned value baseline with a group-relative baseline computed from multiple responses to the same query [21]. For each $x \in \mathcal{D}$, the behavior policy $\pi_{\theta_{\text{old}}}$ samples a group $\{y_i\}_{i=1}^{G}$ and each response receives an outcome $r(x, y_i) \in [0, 1]$. Define the group mean, standard deviation, and standardized advantage:

$$\mu = \frac{1}{G}\sum_{i=1}^{G} r(x, y_i), \quad \sigma = \sqrt{\frac{1}{G}\sum_{i=1}^{G}\big(r(x, y_i) - \mu\big)^2}, \quad A_i = \frac{r(x, y_i) - \mu}{\sigma + \varepsilon}. \tag{2.5}$$

With per-token ratios $w_{i,t} = \frac{\pi_\theta(y_{i,t}|x, y_{i,<t})}{\pi_{\theta_{\text{old}}}(y_{i,t}|x, y_{i,<t})}$, the surrogate mirrors PPO's clipped form while averaging first within a sequence and then across the group:

$$J_{\text{GRPO}}(\theta) = \mathbb{E}\left[\frac{1}{G}\sum_{i=1}^{G}\frac{1}{|y_i|}\sum_{t=1}^{|y_i|}\min\big(w_{i,t}\,A_i,\ \text{clip}(w_{i,t})\,A_i\big)\right] - \beta\,D_{\text{KL}}(\pi_\theta\|\pi_{\text{ref}}). \tag{2.6}$$

GRPO preserves the PPO update logic but substitutes a per-query group baseline for the critic, removing the value model, lowering memory, and simplifying training while keeping the clipped-ratio safeguard and on-policy sampling [21].

Dr. GRPO

Dr. GRPO identifies two biases in the GRPO objective and removes the corresponding normalization terms [12]. First, dividing each sample's loss by $|y_i|$ introduces a response-level length bias. Second, dividing by the group standard deviation σ couples update magnitude to the within-group difficulty spread. Dr. GRPO therefore uses an unnormalized, unbiased group advantage

$$A_i^{\text{Dr}} = r(x, y_i) - \frac{1}{G}\sum_{j=1}^{G} r(x, y_j) \tag{2.7}$$

and removes response-level length averaging:

$$J_{\text{DrGRPO}}(\theta) = \mathbb{E}\left[\frac{1}{G}\sum_{i=1}^{G}\sum_{t=1}^{|y_i|}\min\left(w_{i,t}\, A_i^{\text{Dr}},\ \text{clip}(w_{i,t})\, A_i^{\text{Dr}}\right)\right] - \beta\, D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}}). \tag{2.8}$$

In practice the inner sum is implemented as a masked mean over all tokens in the batch with a constant denominator rather than $|y_i|$ per sample. Removing the two normalizations improves token efficiency and reduces the tendency to inflate the length of incorrect responses [12].

Decoupled Clip and Dynamic Sampling Policy Optimization (DAPO)

DAPO is a large-scale recipe that builds on GRPO with four concrete changes that target entropy collapse, reward noise, and instability [35]. The objective uses asymmetric clipping and aggregates at token level across the whole group:

$$J_{\text{DAPO}}(\theta) = \mathbb{E}\left[\frac{1}{\sum_{i=1}^{G}|y_i|}\sum_{i=1}^{G}\sum_{t=1}^{|y_i|}\min\left(w_{i,t}\, A_i,\ \text{clip}(w_{i,t},\, 1-\epsilon_{\text{low}},\, 1+\epsilon_{\text{high}})\, A_i\right)\right]. \tag{2.9}$$

Here A_i is the group-standardized advantage as in GRPO. Unlike the symmetric clipping used in PPO, GRPO, and Dr. GRPO, DAPO employs asymmetric bounds with $\epsilon_{\text{high}} > \epsilon_{\text{low}}$, allowing stronger positive reinforcement while constraining negative updates more tightly. This asymmetry mitigates training collapse, since excessive negative reinforcement can destabilize the policy. The algorithm removes the KL penalty, uses token-level loss, and adds two stabilizers: *dynamic sampling* with a buffer that enforces a balanced mix of correct and incorrect samples for each query group, and *overlong filtering and shaping* that masks loss on truncated or timed-out generations and downweights overlong failures. Together, the asymmetric clipping, dynamic buffer, and overlong filtering make training robust for very long reasoning traces while preventing early entropy collapse.

Group Sequence Policy Optimization (GSPO)

GSPO [36] addresses critical stability challenges in group-relative policy optimization, particularly for variable-length sequences and Mixture of Experts (MoE) architectures. The algorithm tackles two fundamental problems that limit the robustness of token-level importance weighting. First is aligning the importance weighting to sequence-level advantages. Second is handling small logit discrepancies between training and inference engines that accumulate across long sequences. Token-level importance weighting in GRPO is fragile at scale: the same set of weights operating on the same input sequences may not produce identical logits between training and inference engines due to accumulated floating-point error and differing kernel implementations. In sparse MoE, the logit differences are even more apparent, since the set of activated experts can change between $\pi_{\theta_{\text{old}}}$ and π_θ, making token-level ratios volatile and invalid as off-policy corrections. These effects amplify under clipping and have been observed to trigger irreversible collapse; prior MoE work required routing replay to stabilize GRPO [36]. *GSPO* restores a coherent importance sampler by matching the optimization unit to the reward unit: entire responses. With the same group-relative advantage A_i as in GRPO, define the *length-normalized sequence-level* importance ratio

$$s_i(\theta) = \left(\frac{\pi_\theta(y_i \mid x)}{\pi_{\theta_{\text{old}}}(y_i \mid x)}\right)^{1/|y_i|} = \exp\left(\frac{1}{|y_i|}\sum_{t=1}^{|y_i|}\log\frac{\pi_\theta(y_{i,t} \mid x, y_{i,<t})}{\pi_{\theta_{\text{old}}}(y_{i,t} \mid x, y_{i,<t})}\right), \tag{2.10}$$

and optimize the clipped sequence-level surrogate

$$J_{\text{GSPO}}(\theta) = \mathbb{E}\left[\frac{1}{G}\sum_{i=1}^{G}\min\left(s_i\,A_i,\ \text{clip}(s_i, 1-\epsilon_{\text{low}}, 1+\epsilon_{\text{high}})\,A_i\right)\right]. \tag{2.11}$$

Length normalization in s_i keeps ratios numerically comparable across response lengths and prevents a few token-level logit differences from exploding the sequence weight; the resulting clipping ranges are therefore orders of magnitude smaller than in token-level schemes [36]. Like DAPO, GSPO employs asymmetric clipping bounds to allow stronger positive reinforcement while constraining negative updates. The gradient makes the stability contrast explicit (omitting clipping for brevity):

$$\nabla_\theta J_{\text{GSPO}}(\theta) \;=\; \mathbb{E}\left[\frac{1}{G}\sum_{i=1}^{G} s_i(\theta)\, A_i \;\cdot\; \frac{1}{|y_i|}\sum_{t=1}^{|y_i|} \nabla_\theta \log \pi_\theta(y_{i,t} \mid x, y_{i,<t})\right], \tag{2.12}$$

so all tokens in a response are weighted equally by the same sequence factor $s_i(\theta)$. In GRPO, by contrast, each token carries its own noisy ratio $w_{i,t}$, and those unequal weights accumulate unpredictably along the sequence. GSPO's sequence weighting eliminates that instability, removes the need for routing replay in MoE RL, and tolerates training–inference log-probability precision differences—often allowing direct use of inference-engine likelihoods without recomputation. By default GSPO does not use an explicit KL penalty; when we work with very small effective batch sizes, we sometimes introduce a light KL term purely as a stabilizer and mark this deviation explicitly.

The superior robustness of GSPO makes it particularly attractive for multi-turn agent training where episodes vary substantially in length and structure. By matching the importance weighting granularity to the reward granularity—both at the sequence level—GSPO provides more stable gradients than token-level approaches when training on variable-length trajectories with sparse terminal rewards. These properties led to our adoption of GSPO as the policy optimization algorithm for the Nano agent training described in Chapter 3.

2.5 Low-Rank Adaptation (LoRA)

Parameter-efficient adaptation is essential for training large models without the overhead of full fine-tuning. LoRA learns a low-rank update to frozen weights [5]:

$$\Delta W = BA, \qquad W = W_0 + \frac{\alpha}{r} BA, \tag{2.13}$$

with $B \in \mathbb{R}^{d\times r}$, $A \in \mathbb{R}^{r\times k}$, and rank $r \ll \min(d,k)$. Initializing $B{=}0$ preserves the base model at the start ($W{=}W_0$), which stabilizes optimization. For online Reinforcement Learning with coding agents, LoRA reduces trainable state, supports clean policy-reference separation (adapters differ, base is shared), and accelerates iteration. Adapter placement is flexible—practitioners select which modules to adapt based on capacity/efficiency goals; rank r and scale α set the capacity-efficiency trade-off.

Chapter 3

Method

This chapter presents an execution-free RL methodology for repository-level APR. The approach integrates a minimalist terminal agent (Nano) directly into the training loop, using terminal patch-similarity rewards computed deterministically on canonical diffs to enable language-agnostic learning without executable test harnesses. We first formalize the Nano agent specification (observation space, action space, termination conditions, reward function), then detail training data, policy optimization, and infrastructure requirements. Experimental results addressing the research questions are presented in Chapter 5.

3.1 Overview and Scope

We study whether online RL improves a single minimalist coding agent when grounded in real repositories. The methodology is programming-language agnostic by design: reward depends only on the canonical diff computed from the repository state after the agent's actions, compared against a ground-truth patch. This removes the need for language-specific runners during training and permits controlled data mixing across languages when ground-truth patches exist.

3.2 Training Data

Unlike traditional supervised learning that trains on static datasets, our approach generates training data through interaction with problem instances. Each instance consists of a GitHub repository at a specific buggy commit paired with an issue description and a ground-truth patch. The agent generates episodes by interacting with these repositories through the terminal environment, and training occurs on the trajectories produced from these episodes.

Additionally, repository-level code repair datasets differ fundamentally in how they specify the agent's mission. Instruction-driven datasets provide natural language issue descriptions that communicate what to fix, while test-execution datasets provide failing tests that indicate what is broken. Our execution-free training approach requires the instruction-driven variant.

We employ a 1,000-task curriculum combining Python and multilingual repositories. From SWE-Gym [18], we select 750 Python bug-fixing tasks for training. From SWE-Bench-Multilingual [33], we incorporate 250 multilingual tasks into training, providing coverage across nine programming languages. Due to the sparsity of instruction-driven multilingual data, an additional 50 multilingual instances were held out for potential test-verified evaluation, though time constraints limited the scope to only reward-based analysis presented in Section 5.3. Instruction-driven, repository-level multilingual debugging datasets remain substantially scarcer than their test-execution counterparts, constraining the scale of cross-language training currently feasible.

3.3 Nano Agent

Existing production agent scaffolds optimize for task success through rich tool sets and sophisticated orchestration, but lack properties needed for tractable RL training. Nano fills this gap by prioritizing trainability: unmodified conversation histories that enable end-to-end policy optimization, resource-limit awareness and persistence to lessen reward sparsity, and minimal dependencies for reproducible implementation.

We formalize the Nano agent through its observation space, action space, interaction dynamics, termination conditions, and safety constraints. This specification provides the foundation for the RL training methodology described subsequently.

>NANO

3.3.1 Episode Initialization

Each episode begins with the agent placed within an isolated clone of a Git repository containing a known bug. The agent receives a task prompt in the form of a issue description that specifies the problem to be solved, typically via a bug report. This issue serves as the sole high-level specification of the task; the agent must autonomously navigate the repository, locate relevant code, and implement appropriate fixes using only the provided tools. The repository starts in a clean working state at a specific commit corresponding to the buggy version, and the agent operates entirely through terminal interactions without access to external documentation, pre-computed file listings, or repository metadata beyond what it can discover through exploration.

3.3.2 Observations

All interaction occurs through a terminal transcript maintained as a structured conversation history. The agent observes system messages, issue descriptions, tool outputs, and error messages as they appear in the dialogue. Each tool invocation returns at most 2,000 characters; outputs exceeding this limit are truncated deterministically with an explicit "... output truncated ..." marker appended. The agent must discover all repository information through its own tool invocations; no file tree summaries, test results, or repository statistics are provided beyond what appears in the visible transcript.

3.3.3 Action Space

The agent has two tools: (i) `shell(cmd)` executes within a restricted bash (rbash) with a per-call timeout; and (ii) `apply_patch(path, old_content, new_content)` performs a literal search-and-replace operation without regex. Patches must target files within the current repository; the specified `old_content` must match uniquely in the file at the time of application.

A "null action" terminates the episode when the agent produces no further tool calls: if repository changes exist, this constitutes a submission; if no changes exist, the agent is prompted to continue working until changes are made or limits are reached.

3.3.4 Formal Interaction Loop

The agent operates through a structured cycle that formalizes the relationship between observations, actions, and environmental feedback. At each timestep t, the agent maintains a conversation history h_t comprising the system message, issue description, and all prior agent-environment interactions. Given this history, the model generates an assistant response y_t containing tool invocations expressed as JSON function calls. The inference endpoint extracts the first tool call through a deterministic parser $c_t = \psi(y_t)$, executes it within the isolated repository environment, and returns the resulting output o_t—command results, file contents, or error messages—subject to the 2,000-character truncation policy.

The history advances to $h_{t+1} = \text{append}(h_t, y_t, o_t)$, incorporating both the assistant's generation and the environment's response. This cycle continues until a termination condition is met: the agent submits by producing repository changes without further tool calls, the tool-call budget is exhausted, the token limit is exceeded, or the wall-clock timeout is reached. Upon termination, the episode yields a complete trajectory $\tau = (h_0, y_0, o_0, h_1, \ldots, h_T)$ where T denotes the final timestep.

```
>>> shell(cmd="ls src/")
drwxr-xr-x  utils/
-rw-r--r--  main.py
-rw-r--r--  config.py

>>> shell(cmd="grep -n 'def process' src/utils.py")
42:def process_data(data):
43:    return data.strip().lower()

>>> apply_patch(
...     file_path="src/utils.py",
...     old_content="return data.strip().lower()",
...     new_content="return data.strip().lower().replace(' ', '_')"
... )
Patch applied successfully.
```

Figure 3.3.1: Illustrative episode execution showing the Nano agent's two tools: `shell` for repository navigation and inspection, and `apply_patch` for file modification. The agent explores repository structure through shell commands before applying targeted edits.

3.3.5 Deterministic Canonical Diff Computation

Generating syntactically valid unified diffs presents a high-dimensional formatting challenge for language models: line numbers must accurately reflect current file state, context lines must match existing content exactly character-for-character, and headers must specify correct paths, offsets, and chunk sizes. State-of-the-art models frequently produce malformed diffs with misaligned line numbers, incorrect context, or invalid headers that fail during patch application, introducing a brittle failure mode orthogonal to the semantic debugging task.

The Nano agent sidesteps this issue by using semantically clear search-and-replace operations during interaction and deterministically computing the canonical diff via git after termination, effectively eliminating an entire class of diff formatting errors.

```
$ git diff
diff --git a/src/utils.py b/src/utils.py
index abc123..def456 100644
--- a/src/utils.py
+++ b/src/utils.py
@@ -42,1 +42,1 @@ def process_data(data):
-    return data.strip().lower()
+    return data.strip().lower().replace(' ', '_')
```

Figure 3.3.2: The harness computes the canonical diff via git after episode termination which is subsequently used to compute the reward.

3.3.6 Termination and Limits

Episodes terminate when any of the following conditions is met: (i) the agent produces repository changes and emits no further tool calls (successful submission), (ii) the tool-call budget is exhausted, (iii) the token budget is exceeded, or (iv) the wall-clock timeout is reached. Throughout all experiments, we enforce a maximum of 30 tool calls, a cumulative token budget of 10,240 generated tokens, and a wall-clock timeout of approximately 60 seconds. Individual tool outputs are truncated to 2,000 characters to prevent excessively verbose responses from consuming the token budget. These limits keep Video Random Access Memory (VRAM) utilization bounded and improve sample efficiency: the agent receives warnings as it approaches limits, encouraging it to produce a submission rather than simply timing out.

3.3.7 Sampling and Exploration

Training and evaluation employ distinct sampling configurations that reflect the fundamental exploration-exploitation trade-off in RL. During training episodes, we configure sampling with temperature 1.0 and top-p 1.0 to encourage exploration of diverse tool-use approaches and interaction patterns. This maximally stochastic sampling enables the policy to discover varied solution trajectories and prevents premature convergence to suboptimal behaviors. Conversely, evaluation episodes use temperature 0.2 and top-p 0.9 to exploit the learned policy, producing more deterministic and reliable behavior when assessing performance on held-out benchmarks. This exploration-during-training, exploitation-during-evaluation paradigm ensures that the model explores broadly while learning but executes confidently when deployed.

3.4 Reward Design

The terminal reward $R(\tau) \in [0, 1]$ evaluates the trajectory based on similarity between the canonical diff resulting from the agent's repository modifications and the ground-truth patch.

After an episode terminates, we compute per-file canonical diff hunks for the agent and the ground truth. Let F_a and F_g denote the sets of files modified by the agent and in the ground-truth patch, and let $p_a(f)$ and $p_g(f)$ denote the canonical diff hunk strings for file f. We aggregate per-file similarity and normalize by the greater number of affected files:

$$R(\tau) = \frac{1}{\max\big(|F_a|, |F_g|\big)} \sum_{f \in F_a \cup F_g} \text{similarity}\big(p_a(f), p_g(f)\big) \in [0, 1]. \tag{3.1}$$

Here similarity$(\cdot, \cdot)$ is the string-similarity score of the canonical diff hunks computed with Python's `difflib.SequenceMatcher.ratio()`. Files that appear in only the set of agent affected files or ground truth affected files contribute zero.

This design keeps training simple, avoids brittle intermediate reward engineering, and permits diverse problem-solving trajectories while targeting the ultimate objective directly: producing the correct changes to the repository. By rewarding patch similarity rather than test passage, this execution-free approach enables language-agnostic training without language-specific test infrastructure, build systems, or execution environments. The trade-off exchanges direct functional verification for infrastructure simplicity and determinism, making large-scale multilingual training tractable.

3.5 Policy Optimization

We adopt GSPO [36] as our policy optimization algorithm. As detailed in Section 2.4.3, GSPO extends GRPO with sequence-level importance weighting, offering superior stability for variable-length episodes compared to token-level methods—critical for multi-turn agent training where episodes vary substantially in length and structure.

To further stabilize training under small effective batch sizes constrained by available compute, we add a light KL regularizer to the initial pre-trained checkpoint. Implementation details including batch configuration, custom Triton kernels for GSPO loss computation, and gradient masking for multi-turn trajectories are documented in

Chapter 4.

3.6 Model Choice

We base our experiments on Qwen3-14B, a hybrid reasoning model combining strong coding capabilities with native tool-calling support. Qwen3 was selected based on widespread consensus that it was the strongest open-weight coding model for tool-augmented tasks at the time of experimentation. The 14B variant offered an optimal fit for available resources: Qwen3-8B proved too large for two GPUs yet too small to justify three, while the larger 32B variant imposed prohibitive computational demands.

Several alternative models including Llama3.1-8B, Gemma3, Ministral, GLM4-9B, and Llama3.1-Nemotron were explored but ultimately excluded. Limited comparisons across Qwen3 sizes and Llama3.1-8B, along with more detailed justification of model selection constraints, appear in Chapter A.

Chapter 4

Infrastructure: Live Weight Synchronization for Online RL

Chapter 3 established the methodology: execution-free patch-similarity rewards, the Nano agent specification, GSPO training, and evaluation protocol. This chapter documents the practical realization of that methodology, addressing two primary challenges: enabling online RL training with live policy updates to deployed inference servers, and making multi-turn agent training tractable within academic compute constraints through systematic optimization.

We detail the training environment implementation, including live adapter synchronization via NCCL, trainer extensions for variable-turn episodes, and orchestration under SLURM. Compute-efficiency optimizations compose parameter-efficient adaptation, memory management, gradient checkpointing, mixed-precision training, and fused kernels to enable 14B parameter training on modest GPU clusters. Complete implementation artifacts and configurations appear in Chapter C.

4.1 Online RL Infrastructure

Recent RL training libraries have begun supporting out-of-process generation, where episode collection runs on separate inference servers rather than within the trainer process itself. For most use cases, vLLM's synchronous v1 engine suffices: the trainer requests a batch of prompts, waits for completion, and processes the returned sequences. This approach works well for single-turn generation but proves inefficient for multi-turn coding agents, where existing methods execute each turn synchronously across the entire batch.

We adopt a different architecture that leverages vLLM's asynchronous engine and its OpenAI-compatible API. The async engine implements continuous batching, tool call parsing, and KV-cache management necessary for multi-turn agent interactions. From the trainer's perspective, we provide a method that synchronously returns a batch of completed episode trajectories. Internally, however, individual agents progress through their multi-turn interactions asynchronously, substantially improving throughput. The batch remains synchronous at the episode level to satisfy group-relative policy optimization requirements.

The OpenAI-compatible API interface decouples the AI application logic from trainer abstractions. The agent runs identically during training and deployment—the same server configuration, tool definitions, and interaction logic in both contexts. The trainer operates only on issue descriptions and completed trajectories, requiring no agent-specific code.

Figure 4.1.1 illustrates the training loop. After the trainer collects N completed trajectories and updates the policy, modified weights synchronize to the vLLM server via NCCL without restarting the inference service.

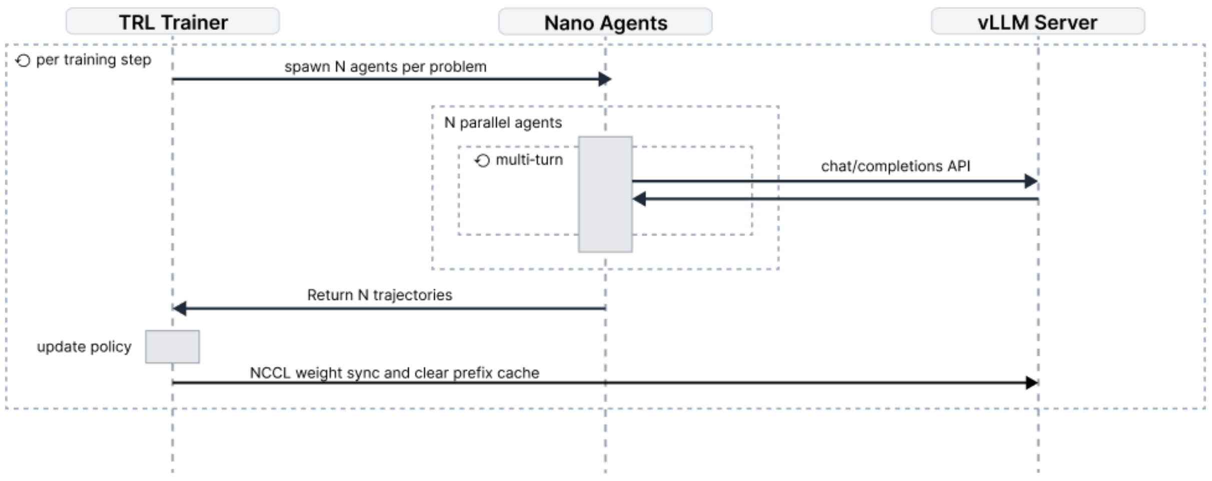

Figure 4.1.1: Training loop architecture showing asynchronous episode collection through the vLLM server's API and subsequent weight synchronization via NCCL.

Standard vLLM loads weights at server initialization and keeps them frozen thereafter. Our key technical contribution extends the async engine to accept NCCL-based weight updates from the distributed trainer, enabling continuous policy improvement without service restarts.

4.2 Live Weight Synchronization

Weight synchronization requires extensions on both sides of the training-inference boundary. We inject a worker into vLLM's async engine that manages an NCCL communicator and exposes parameter update and kv-cache resetting operations, while the trainer instantiates a corresponding client. At system initialization, the trainer launches on designated training GPUs and waits for the NCCL handshake, while the vLLM server launches on separate inference GPUs and opens the communicator, establishing a persistent point-to-point channel. This architecture supports flexible deployment topologies, accommodating co-located processes on a single node or distributed configurations across networked infrastructure. Training commences once the communication channel initializes.

After each policy update, the trainer gathers updated parameters sequentially, layer by layer, from the distributed training nodes. For LoRA-adapted models, adapters are merged into their base layers before transmission, rendering the synchronization mechanism agnostic to whether training uses full weights or parameter-efficient adaptation—the inference server receives standard weight tensors in both cases. This layer-wise gathering bounds peak memory consumption during the collection phase while the sequential transmission introduces negligible latency relative to network transfer time.

The gathered weights broadcast to the inference server, where the worker process updates model parameters layer by layer. After parameter updates, we invalidate all cached key-value states, as cached projections were computed using previous weights. The worker resets the prefix cache completely, ensuring subsequent requests execute with consistent weights and cache state.

The system coordinates synchronization timing such that no episode requests arrive during the weight update and cache reset window. This design eliminates race conditions between parameter updates and ongoing inference without requiring explicit locking mechanisms or request queuing—the training loop naturally gates new episode dispatch on synchronization completion.

4.3 Compute Optimizations

A central objective of this work was making online RL for coding agents tractable within academic compute budgets. A naive implementation of our training pipeline quickly exhausts available VRAM: a 14B parameter model consumes 28GB in BF16 precision for weights alone, before accounting for optimizer states (another 28GB for AdamW), gradients, or the substantial activation memory accumulated across long multi-turn episodes. Training on three A100 GPUs rather than industrial clusters with dozens of accelerators required addressing multiple simultaneous memory and computation bottlenecks.

Aside from standard optimizations like BF16 mixed-precision training and FlashAttention kernels, we describe key techniques that address distinct bottlenecks and compose synergistically.

4.3.1 Limit Aware Agent Design for Improved Reward Signal

Nano imposes explicit episode limits that serve dual purposes: bounding VRAM utilization and improving sample efficiency. Token limits prevent out-of-memory failures as conversation histories grow, while tool-call budgets and wall-clock timeouts prevent runaway episodes from monopolizing training resources. Rather than aggressively truncating episodes to fit memory constraints—which risks discarding tokens that influenced the final reward—we set budgets that accommodate realistic debugging workflows. The agent receives warnings when approaching limits, encouraging submission attempts that ensure episodes yield gradient information even for unsolved tasks.

4.3.2 LoRA for Parameter-Efficient Adaptation

We restrict updates to low-rank adapter matrices (rank 32, $\alpha = 64$) applied to attention and Multi-Layer Perceptron (MLP) projections, vastly reducing the amount of trainable parameters. This choice yields compounding benefits: optimizer states shrink proportionally, reference policy access for KL computation requires only unmounting adapters rather than loading duplicate weights, and weight synchronization transmits compact adapter deltas rather than the full 28GB model. The restricted parameter space also improves training stability by avoiding simultaneous updates to billions of pre-trained parameters.

4.3.3 Custom GSPO Triton Kernels for Loss Computation

The GSPO loss involves multiple sequential operations—log-probability computation, masking, importance weighting, advantage normalization, and sequence-level clipping—that materialize intermediate tensors when implemented as separate PyTorch operations. Following design principles from Liger-Kernel, we developed fused Triton kernels that combine these operations into single GPU kernels, eliminating intermediate materialization. Liger-Kernel reports 60% memory reduction and 20% throughput gains for core training primitives, while TRL documents 40% peak memory reduction during GRPO training with fused kernels.

4.3.4 DeepSpeed ZeRO for Optimizer State Sharding

We employ DeepSpeed ZeRO-2 partitioning, which shards optimizer states and gradients across training GPUs while replicating model parameters. This configuration addresses the primary memory bottleneck: AdamW optimizer states (first and second moments) would otherwise double the 28GB parameter memory footprint, exceeding per-device VRAM budgets. We choose ZeRO-2 over ZeRO-3 to avoid parameter-gathering latency during forward and backward passes, as 14B model parameters fit comfortably when replicated across two A100s.

4.3.5 Gradient Checkpointing for Activation Memory

Long multi-turn episodes accumulate substantial activation memory during backward passes. We employ gradient checkpointing at transformer layer boundaries, discarding intermediate activations during forward passes and recomputing them on demand during backpropagation. This typically reduces peak activation memory by 40-60% while adding approximately 20-33% compute overhead.

4.3.6 Gradient Accumulation for Effective Batch Size

Computing group-relative advantages requires that an entire group fit in memory simultaneously; in our configuration we set $G=8$. With two training GPUs, we set a per-device batch size of 4 episodes, fully utilizing available VRAM given the long context lengths of multi-turn debugging episodes. To achieve higher effective batch sizes without exceeding memory constraints, we accumulate gradients across 4 mini-batches of 8 episodes before applying optimizer updates. This configuration ensures that 32 episodes contribute to each policy update, trading increased wall-clock time per training step for improved gradient stability through larger effective batch sizes.

Chapter 5

Experimental Results

This chapter presents experimental results addressing the three research questions established in Chapter 1. We begin by validating training convergence across diverse programming languages (RQ1), then examine scaffold adaptation (RQ2) and SWE-Bench-Verified performance (RQ3), concluding with a discussion of the observed patterns and their implications.

5.1 RQ1: Training Convergence Across Languages

Research Question: Does execution-free GSPO training converge across diverse programming languages?

Approach: We monitor training dynamics over 460 gradient steps spanning 2 epochs of the 1,000-task curriculum (750 Python + 250 multilingual tasks across nine languages). Overall convergence is assessed through mean and standard deviation of patch-similarity rewards computed after each episode, with reward progression indicating learning and sustained variance ensuring strong gradient signal. Language-level convergence is validated by comparing per-language rewards on identical problem instances between epoch 1 (first exposure) and epoch 2 (second exposure). For group-relative methods like GSPO, sustained variance is critical: when all responses receive similar rewards, advantages collapse toward zero and eliminate the policy gradient.

5.1.1 Overall Training Dynamics

Training proceeded for a little over 2 epochs, completing in 2 days of wall-clock time on 3 A100 GPUs (144 total GPU-hours), with Figure 5.1.1 demonstrating sustained learning throughout. Mean patch-similarity rewards approximately doubled from 0.05 to peak values exceeding 0.10. The continued upward trend validates that execution-free rewards provide sufficient signal for policy optimization in the agentic debugging setting.

Crucially, reward standard deviation shows steady increase from 0.02 to 0.07 rather than collapsing toward zero. In group-relative optimization methods such as GSPO, advantages are computed by comparing each response's reward against within-group statistics that approximate the value function. When all responses receive similar rewards, this approximation becomes degenerate: uniform outcomes provide no signal for distinguishing better from worse actions, causing advantages to approach zero and eliminating the policy gradient. Moreover, the variance itself enables learning the value function approximation—without diversity in outcomes, the baseline cannot meaningfully estimate expected returns. The sustained and growing variance indicates the policy generates diverse episode outcomes with meaningfully different rewards, maintaining strong policy gradient signal throughout training. This stands in contrast to variance collapse, a common failure mode where the policy converges prematurely to a narrow strategy that produces uniform outcomes.

The transient reward decline around step 200 reflects a characteristic challenge in online reinforcement learning. As detailed in Section 5.2, the tool usage patterns that proved effective through step 150 subsequently degraded performance, and it took considerable time for alternative patterns to reemerge in the policy distribution. These alternative patterns ultimately proved superior, yielding the subsequent reward improvements. Similar transitions occur throughout training as the policy sifts through different behavioral patterns.

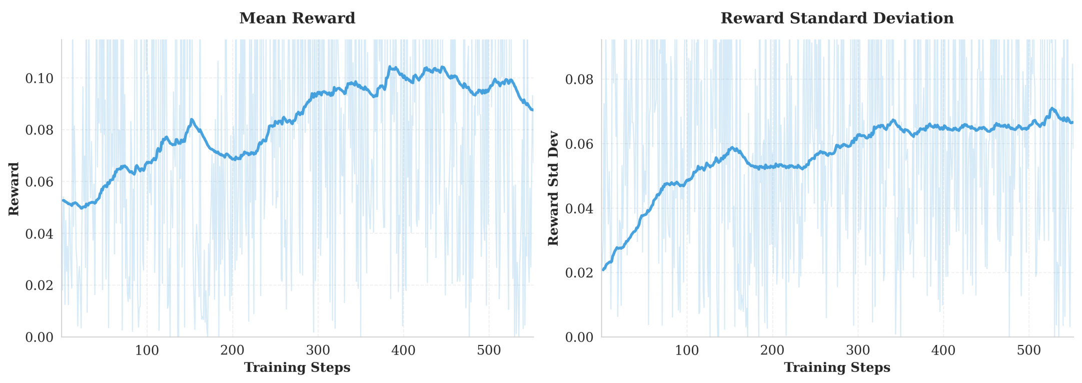

Figure 5.1.1: Mean patch-similarity reward (left) and standard deviation (right) throughout training, plotted with exponential moving average smoothing ($\alpha = 0.01$). Sustained variance growth indicates the policy maintains strong gradient signal by generating diverse episode outcomes, avoiding variance collapse.

5.1.2 Language-Level Convergence

The overall convergence demonstrated above aggregates performance across all languages in the training curriculum. To validate that execution-free rewards enable effective learning across diverse programming ecosystems, Figure 5.1.2 presents per-language reward progression from epoch 1 to epoch 2. All nine languages demonstrate measurable learning progression despite substantial differences in syntax, semantics, and runtime characteristics, including systems languages (C, Rust), dynamically-typed languages (Python, Ruby, JavaScript, PHP), and statically-typed languages (Java, Go, TypeScript).

Despite substantial variation in absolute reward magnitudes (likely reflecting language verbosity differences and task difficulty), every language shows epoch-over-epoch improvement. This validates that patch-similarity rewards provide effective learning signals across diverse syntax and semantics without language-specific adaptations, confirming that execution-free training supports unified policy optimization across heterogeneous language ecosystems.

5.2 RQ2: Nano Agent Scaffold Adaptation

Research Question: How does GSPO training improve Nano agent scaffold adaptation?

Approach: We analyze operational metrics extracted from training episode logs:

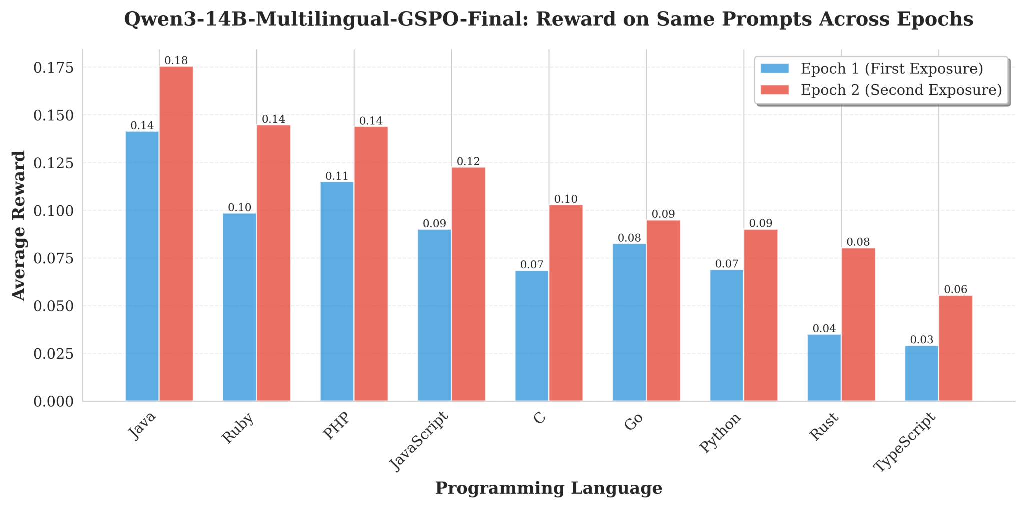

Figure 5.1.2: Per-language reward progression from epoch 1 to epoch 2 on the same problem instances, demonstrating convergence across nine programming languages.

tool execution success rates (proportion of syntactically valid tool calls that execute without errors), episode length (number of tool calls per episode), and command usage patterns (frequency distribution of specific shell commands over time). All metrics are computed over the training trajectory and visualized with exponential moving average smoothing ($\alpha = 0.05$) to reveal trends. These operational measures characterize how the policy's scaffold interaction behavior evolves during training, complementing the reward progression analysis in Section 5.1.

5.2.1 Episode Length

Figure 5.2.1 tracks the number of tool calls per episode throughout training. Initially, the policy consistently exhausts the maximum episode budget of 30 tool calls, frequently exhibiting "death spiral" behavior where it repeatedly invokes the same failing tool without adapting its strategy. The reward decline around step 200 discussed in Section 5.1 coincides with a spike in apply_patch usage visible in Figure 5.2.3.

As training progresses, the policy ceases to exhaust the tool call budget and episode lengths trend consistently downward. This indicates that the policy learns both to execute successful debugging workflows more efficiently and to terminate episodes when productive actions are exhausted rather than persisting with unproductive patterns.

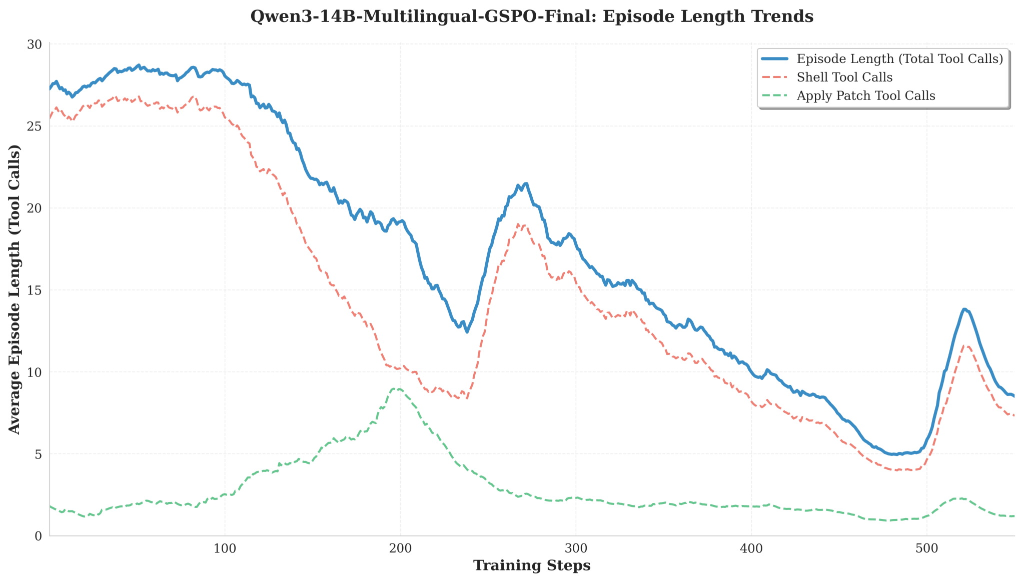

Figure 5.2.1: Number of tool calls per episode throughout training, plotted with exponential moving average smoothing ($\alpha = 0.05$).

5.2.2 Tool Execution Success Rates

Tool execution success measures whether the policy generates syntactically valid JSON tool calls that execute without errors. As noted in Section 5.2.1, episodes shorten substantially over training, which could artificially inflate success rates if the policy simply avoids aforementioned death-spirals. However, success rate improvements occur throughout training as rewards increase, including during the early phase when episodes consistently exhaust the 30-call budget, indicating that the policy learns to generate more valid tool calls rather than merely becoming more selective about which tools to invoke.

Figure 5.2.2 shows tool execution success rates for both tools throughout training. The shell command success rates improve from approximately 45% early in training to 80% at the end, though the peak success rate coincides with episodes averaging only 5 tool calls. The apply_patch success rates exhibit greater volatility, but still trend upwards.

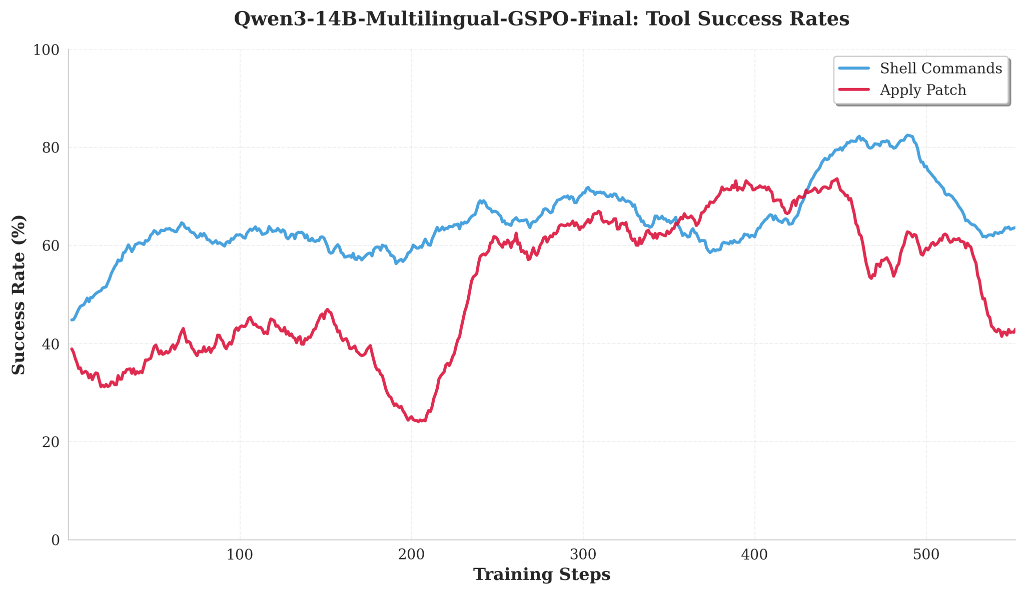

Figure 5.2.2: Tool execution success rates throughout training, computed as the mean proportion of successful invocations per tool type across episodes. Plotted with exponential moving average smoothing ($\alpha = 0.05$).

5.2.3 Command Usage Evolution

While Section 5.2.1 examined overall shell command and apply_patch counts, Figure 5.2.3 decomposes shell command usage into the most frequently invoked individual commands to reveal strategic shifts in debugging behavior. Each command's usage is expressed as a percentage of total tool calls at that training step, allowing us to broadly observe the patterns the policy favors as a function of time.

5.3 RQ3: SWE-Bench-Verified Performance

Research Question: Does execution-free patch-similarity RL training improve SWE-Bench-Verified performance?

Approach: We evaluate on the full 500-instance SWE-Bench-Verified benchmark using identical evaluation infrastructure (Nano scaffold, inference settings, episode limits) for both baseline and post-training checkpoints. The baseline is Qwen3-14B operating through the Nano scaffold before any RL training. Primary metrics are completion rate (episodes producing non-empty repository modifications) and resolve rate (test-verified functional correctness). Both evaluations use deterministic generation to isolate policy learning effects from sampling variance.

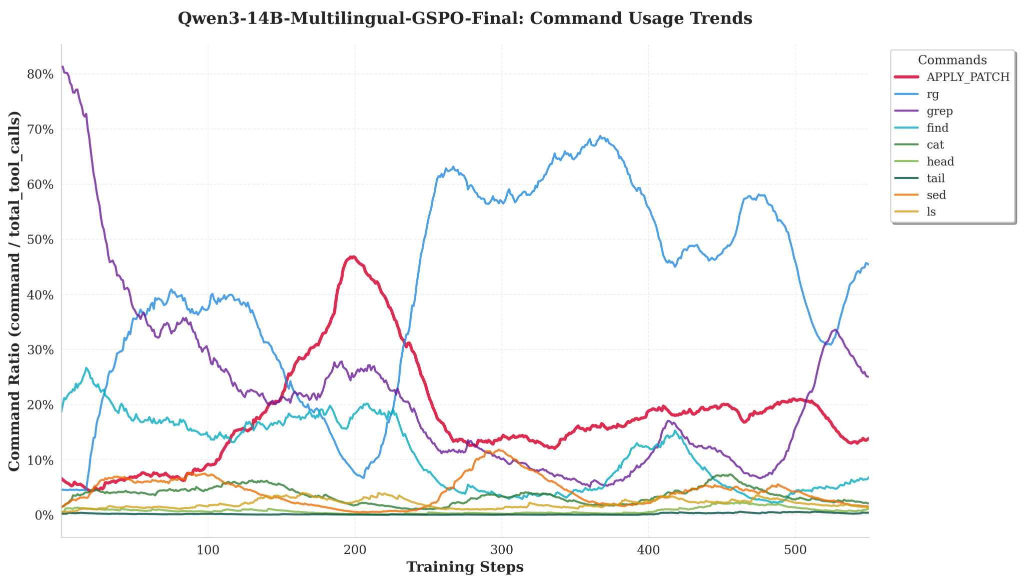

Figure 5.2.3: Evolution of command usage frequencies throughout training, plotted with exponential moving average smoothing ($\alpha = 0.05$).

Table 5.3.1 compares SWE-Bench-Verified performance before and after GSPO training. The baseline model, evaluated using the identical Nano agent before RL training, demonstrates substantial difficulties with fundamental agent operation. Although all 500 instances receive submission attempts, only 186 episodes (37.2%) produce non-empty patches, while 313 episodes (62.6%) fail to modify the repository. This high empty-patch rate reveals that the pre-trained model struggles to execute valid debugging workflows within episode constraints, frequently exhausting token budgets or tool-call limits without applying repository changes. Among the 186 episodes that produce patches, 36 instances (7.2% of total) pass all associated test cases.

After GSPO training at step 460, episodes producing patches more than double to 388 instances (77.6%), reducing empty-patch failures by 64%. This demonstrates substantial improvement in scaffold adaptation: the trained model reliably executes multi-turn debugging episodes, navigates repositories, and applies patches within operational constraints.

However, test-verified success decreases from 36 to 31 resolved instances (7.2% $\rightarrow$ 6.2%). Given the small absolute numbers and inherent evaluation variance, this difference likely falls within error margins and may reflect statistical noise rather than meaningful degradation. Nonetheless, the results clearly indicate that training has not yet improved functional correctness despite clear gains in operational reliability.

Table 5.3.1: SWE-Bench-Verified performance comparing pre-training baseline and post-GSPO training on 500 instances. The baseline model is Qwen3-14B with the Nano agent before RL training. All metrics are presented such that higher values indicate better performance.

Metric	**Baseline**	**Ours (step 460)**
Completed instances	186	**388**
Completion rate	37.2%	**77.6%**
Resolved instances	**36**	31
Resolve rate	7.2%	6.2%

While test-verified success remains flat, mean patch-similarity rewards on these same SWE-Bench-Verified submissions increase substantially. Table 5.3.2 reports performance on the training objective itself: textual similarity between generated patches and ground-truth modifications.

Table 5.3.2: Mean patch-similarity rewards on SWE-Bench-Verified instances.

Metric	**Baseline**	**Ours (step 460)**
Mean reward	0.122	**0.189**

The 54% reward improvement demonstrates that the policy is learning according to the training objective (patch similarity), even though this learning has not yet translated to improved test-verified success rates.

5.4 Discussion

Training demonstrates clear convergence: rewards double, variance sustains, operational metrics improve substantially, and learning generalizes across nine programming languages. Yet test-verified success on SWE-Bench-Verified remains flat at approximately 6–7%.

One plausible explanation involves the initial reward distribution, where 313 of 500 instances (62.6%) produce empty patches. This creates a natural learning progression: escaping the zero-reward regime offers abundant signal where any valid patch constitutes measurable progress, while refining patch quality toward higher similarity scores becomes feasible only after the policy reliably produces non-zero outcomes. Step 460's trajectory aligns with this interpretation. Completion rates increase from 37% to 78%, tool success improves from 45% to 80%, and the 54% reward gain demonstrates operational competence, but this competence has not yet translated to test-verified improvements. The transient reward decline at step 200, coinciding with increased apply_patch usage, may reflect temporary overweighting of patch production at the expense of quality.

Extended training may bridge this gap, particularly given that similar methodologies (DeepSWE [14]) achieve strong results after substantially longer training with greater resources. Alternative explanations remain possible, including fundamental limitations of patch-similarity rewards for functional correctness, though continued upward reward trends suggest the policy has not exhausted learning from the current objective.

The results validate core feasibility: execution-free RL training converges across diverse programming languages and drives measurable scaffold adaptation. The gap between operational competence and functional success highlights open questions about training duration, reward design, and the relationship between textual similarity and semantic correctness.

Chapter 6

Conclusions and Future Work

This thesis investigates online RL training for repository-level code repair through a single minimalist agent, demonstrating that execution-free patch-similarity rewards enable meaningful improvements in scaffold adaptation within academic compute constraints. We developed infrastructure for live weight synchronization between distributed trainers and deployed inference servers, enabling efficient multi-turn agent training on modest GPU clusters. Training Qwen3-14B with GSPO on execution-free rewards substantially improves operational reliability, more than doubling patch completion rates from 37% to 78%, though test-verified correctness remains flat at approximately 6–7%.

6.1 Summary of Contributions

Our work makes several methodological and infrastructure contributions to online RL training for coding agents:

6.1.1 Execution-Free Training Methodology

Language-Agnostic Rewards: We demonstrate that execution-free patch-similarity rewards enable effective RL training across programming languages without language-specific execution infrastructure. Training on a 1,000-task curriculum mixing Python and nine additional languages validates that static diff comparison provides sufficient learning signal for repository-level code repair.

Nano Agent Design: The minimalist Nano agent operates with only two tools, restricted bash and search-and-replace patching, demonstrating that operational agent competence can develop through RL training without engineered complexity. By sidestepping diff generation complexity through deterministic git-based canonical diff computation, the design eliminates an entire class of formatting errors orthogonal to the semantic debugging task.

6.1.2 Technical Infrastructure

Live Weight Synchronization via NCCL: We extend vLLM's async engine to accept live parameter updates from distributed trainers through persistent NCCL channels, enabling continuous policy improvement without server restarts. The implementation handles layer-wise parameter gathering, LoRA adapter merging before transmission, and complete KV-cache invalidation after updates, maintaining consistency between weights and cached activations. This eliminates the conventional training-then-deploy cycle, enabling true online learning where policy updates immediately propagate to deployed inference infrastructure.

Asynchronous Multi-Turn Architecture: By leveraging vLLM's OpenAI-compatible API, we enable individual agents to progress through multi-turn interactions asynchronously while maintaining synchronous batches at the episode level for group-relative optimization. This decouples AI application logic from trainer abstractions, ensuring the agent runs identically during training and deployment without requiring agent-specific code in the training loop. The architecture substantially improves throughput over synchronous per-turn execution while satisfying GSPO batch requirements.

Academic Feasibility Optimizations: Through composed optimizations including LoRA parameter-efficient adaptation, DeepSpeed ZeRO-2 for optimizer state sharding, gradient checkpointing for activation memory, and custom Triton kernels for GSPO loss computation, we demonstrate 14B parameter training on three A100 GPUs with two for training and one for inference. These techniques compose synergistically to address distinct memory and computation bottlenecks, making online RL for multi-turn agents tractable within typical academic allocations.

Open-Source Release: Complete training infrastructure, agent implementations, SLURM orchestration scripts, and evaluation protocols are released open-source, enabling reproducible investigation of online RL techniques and reducing barriers to academic research in experiential learning for coding agents. By providing complete open-source implementations, we enable broader academic investigation of online RL techniques for coding agents, facilitating reproducible research and community-driven development of advanced agent training methods.

6.1.3 Sustainability

The compute optimization techniques detailed in Chapter 4 directly address sustainability across environmental, economic, and social dimensions. By enabling 14B parameter online RL training on three A100 GPUs rather than industrial-scale clusters, we reduce both absolute energy consumption and the concentration of research capabilities in well-funded institutions. Training completed in 144 GPU-hours, demonstrating that meaningful RL research remains feasible within academic compute budgets and associated energy constraints. The execution-free reward design eliminates the operational overhead of maintaining language-specific test runners, build systems, and Docker orchestration across nine programming languages, further reducing both engineering complexity and computational requirements. From an economic perspective, the complete open-source release of training infrastructure, agent implementations, and orchestration scripts (Chapter C) creates a sustainable research ecosystem where advanced techniques remain accessible beyond proprietary laboratories, enabling sustained academic investigation rather than dependence on selective corporate disclosures. This low-resource methodology promotes equitable research participation: by targeting modest GPU allocations typical of academic clusters and adopting language-agnostic designs that avoid complex per-language execution infrastructure, we reduce barriers that would disproportionately burden under-resourced institutions attempting to replicate or extend this work.

6.1.4 Ethics

Autonomous code repair systems capable of repository modification raise several ethical considerations that warrant careful discussion. While developed for beneficial bug repair, similar capabilities could theoretically facilitate automated vulnerability exploitation or malicious code injection, presenting dual-use concerns inherent to any autonomous system operating on software artifacts. We mitigate immediate operational risks through restrictive execution environments during training, including restricted bash, per-episode tool-call limits, and wall-clock timeouts, though we acknowledge these measures target training safety rather than preventing determined misuse of released models. Training data exclusively uses permissively licensed repositories from SWE-Gym and SWE-Bench-Multilingual, ensuring legal and ethical sourcing while avoiding exposure of proprietary code or credentials, and the execution-free approach prevents accidental leakage of secrets that might occur during test execution. Production deployments would require stronger isolation than our SLURM-compatible approach provides; we document these limitations transparently to inform responsible deployment decisions. The complete open-source release embodies a commitment to research transparency: by publishing training recipes, infrastructure code, and evaluation protocols, we enable independent verification and community-driven safety research, contrasting with proprietary development where capabilities and failure modes remain opaque.

6.2 Limitations and Constraints

The methodology presented here operates within several constraints that bound its immediate applicability and suggest directions for future refinement. While execution-free training converges reliably and improves operational competence substantially, the approach reflects deliberate trade-offs between performance, generality, and resource requirements.

6.2.1 Performance Levels and Task Scope

While GSPO training substantially improved scaffold adaptation, more than doubling patch completion rates from 37% to 78%, test-verified success on SWE-Bench-Verified remained flat at approximately 6-7%. This gap likely reflects insufficient training duration rather than fundamental methodological limitations. DeepSWE [14], which applies similar online RL methods to terminal-based coding agents, demonstrates strong test-verified performance after training with more than 30x computing power for 3x longer. Given the methodological similarities, replicating such results may primarily require additional training rather than architectural changes.

The evaluation focuses on single-commit bug fixes rather than complex architectural changes or multi-file refactoring tasks. Repository-level changes spanning dozens of files with intricate dependency relationships present substantially harder challenges that may require different agent designs or training methodologies.

6.2.2 Task Distribution and Language Coverage

Training data combines 750 Python tasks with 250 multilingual tasks spanning nine additional languages. This distribution reflects dataset availability: instruction-driven, repository-level multilingual debugging datasets remain substantially scarcer than Python-only resources. Multilingual generalization is assessed through per-language reward progression across training epochs, demonstrating improvements across all languages despite the uneven data distribution.

6.2.3 Computational Resource Requirements

Online RL training requires substantially more computation than supervised fine-tuning: each training example involves multi-turn agent interaction rather than single forward-backward passes. While our optimizations make training tractable on three A100 GPUs, the complete system integrating distributed training, live inference serving, weight synchronization infrastructure, and episode orchestration presents implementation complexity that may challenge researchers without prior distributed training experience. The open-source release includes comprehensive documentation and SLURM orchestration scripts to mitigate these barriers.

6.2.4 Execution Environment Isolation

Episodes execute in ephemeral repository checkouts with restricted bash (rbash) rather than full containerization. This approach provides sufficient isolation for safe training and reproducible execution while remaining simpler to deploy on HPC clusters where container runtimes may require special permissions. However, container-based isolation would enable higher parallelism by safely running more concurrent episodes per node, substantially increasing throughput at the cost of additional infrastructure complexity.

6.2.5 Reward Function Trade-offs

Execution-free patch-similarity rewards trade functional verification for infrastructure simplicity and language-agnostic applicability. Static diff comparison may occasionally reward syntactically plausible but functionally incorrect patches, or penalize alternative valid solutions that achieve the same functional outcome through different code changes. Conversely, test-based rewards would require maintaining language-specific execution environments, test runners, build systems, and dependency management for each programming language, substantially increasing engineering complexity. The trade-off favors execution-free rewards for academic research contexts prioritizing broad language coverage and rapid experimental iteration.

6.2.6 Model Architecture and Scale

This work focuses on the Qwen3 model family (primarily 14B parameters) due to strong tool-calling capabilities and computational accessibility within academic budgets. The methodology's effectiveness across models with different architectural characteristics, such as varying attention mechanisms, tokenization schemes, or pre-training objectives, requires validation beyond the limited multi-size experiments documented in Chapter A. Whether the observed improvements generalize to other model families or depend on Qwen3-specific capabilities remains an open empirical question.

6.3 Future Research Directions

The infrastructure and methodology developed in this thesis open several promising directions for extending online RL training of coding agents. We identify four areas where architectural refinements, expanded scope, or increased scale could yield substantial improvements in both system efficiency and agent capabilities.

6.3.1 Cross-Group Walltime Optimization

Our current implementation optimizes intra-group asynchronous behavior, allowing individual episodes within a training group to progress independently. However, the system treats episode groups synchronously: inference pauses once the configured group size is reached, waiting for training to complete and weights to synchronize before new episodes begin. Given the substantial variation in episode completion times, where some problems resolve in five turns while others require twenty or more, this synchronous grouping creates walltime dependencies where inference servers sit idle during training updates.

Future implementations could eliminate this bottleneck through slightly off-policy episodes. Instead of pausing inference during training, new episodes would begin immediately using the current policy while training proceeds on completed episodes in parallel. When weight updates broadcast to inference servers, any in-flight episodes would continue with the updated policy, making them slightly off-policy since they began with policy version π_n but will complete under π_{n+1} or later. Given the small, gradual nature of GSPO updates and continuous weight synchronization, this off-policy gap remains bounded and acceptable.

This approach would maintain continuous inference server utilization, eliminate idle time during training phases, and enable training to begin as soon as any subset of episodes completes rather than waiting for full groups. The optimization would particularly benefit throughput when episode length distributions exhibit high variance, as faster episodes immediately feed into training while longer episodes continue without blocking progress.

6.3.2 Unified Training-Inference Architecture

The current system separates training and inference across distinct GPU pools with explicit weight synchronization. While this design maximizes throughput by overlapping computation, it increases infrastructure requirements: even modest experiments demand three GPUs (two for training, one for inference). Future implementations could co-locate training and inference on a single set of weights, eliminating synchronization overhead entirely and reducing the entry barrier for RL experimentation.

In this unified architecture, the model would alternate between inference and training phases rather than running them concurrently. Episodes would accumulate using the current policy weights, and once sufficient experience is collected, the system would pause episode generation, perform training updates in-place, then resume inference with the updated policy. This eliminates the need for separate inference servers, weight broadcasting infrastructure, and NCCL synchronization—reducing the minimum viable system to a single training node.

The trade-off is straightforward: training updates pause inference, reducing overall throughput compared to the parallel design. However, for researchers with limited compute budgets or those conducting initial feasibility studies, this reduced throughput may be acceptable in exchange for dramatically lower infrastructure complexity and resource requirements. The design would particularly benefit early-stage experimentation where validating core concepts matters more than maximizing sample efficiency.

6.3.3 Multi-Task Curriculum Training

The OpenAI-compatible API abstraction and plug-and-play harness design naturally extend to multi-task training scenarios. Rather than optimizing a single objective (code repair), the system could train concurrently on diverse tasks—code repair, email classification, document summarization, or any other problem admitting a structured reward signal. Episodes from different task distributions would interleave within training groups, and the shared policy would learn to handle heterogeneous environments through the same tool-calling interface.

This multi-task formulation raises fundamental questions about capability emergence and transfer in RL post-training. Does sharing model capacity across diverse objectives improve generalization, allowing debugging skills to transfer to related reasoning tasks? Or does multi-task training introduce interference, degrading performance on individual objectives compared to specialized single-task policies? The answers would illuminate whether RL-trained LLMs benefit from curriculum diversity analogous to pre-training's broad data mixtures, or whether task-specific specialization remains necessary for complex reasoning domains.

Investigating these questions could reveal fundamental insights about the future of RL post-training: whether general-purpose reasoning emerges from diverse experiential curricula, or whether different cognitive skills require isolated optimization. The infrastructure documented in this thesis provides the foundation for such investigations, requiring only task-specific harnesses and reward functions to explore multi-objective training systematically.

6.3.4 Extended Training with Scaled Resources

Several lines of evidence suggest that extended training with moderately larger models would yield stronger results. DeepSWE demonstrates that similar online RL methodologies achieve strong test-verified performance with longer training duration and more computational resources. Our reward curves continue trending upward throughout training without plateauing, indicating that the policy has not exhausted learning from the patch-similarity signal. Additionally, preliminary comparisons in Chapter A show that Qwen3-32B achieves substantially higher baseline performance than the 14B variant, reflecting that RL effectiveness scales with underlying model capability. Future work should investigate training larger models for extended durations while maintaining the low-resource optimization techniques documented in Chapter 4. The infrastructure supports 32B parameters on 6 A100s, enabling investigation of whether the flat test-verified success rates reflect model capacity and training duration constraints rather than fundamental methodological limitations.

6.4 Lessons Learned from Implementation

The process of implementing and deploying online RL for coding agents yielded insights that extend beyond the specific technical contributions documented in this thesis. These reflections illuminate both the opportunities and challenges inherent in experiential learning approaches for complex reasoning tasks.

6.4.1 System Complexity and Engineering Effort

Integrating large language models with online RL training revealed engineering challenges that significantly exceeded initial estimates.

Many critical failure modes only emerged during extended training runs spanning multiple days, revealing the difficulty of predicting distributed system behavior through local testing. Robust monitoring infrastructure and iterative refinement proved essential—ambitious initial designs consistently required substantial simplification before achieving stability.

Engineering effort for production-ready implementation substantially outweighed theoretical development. Implementing weight-synchronized OpenAI-compatible API servers required many weeks of trial and error to achieve stable operation, revealing that infrastructure reliability demands sustained investment beyond algorithmic innovation.

Integrating bleeding-edge libraries amplified this challenge. The system relied on rapidly evolving dependencies including TRL for RL training, vLLM for inference serving, Liger-Kernel for memory-efficient kernels, and Flash Attention for optimized attention, each offering significant performance benefits but introducing breaking changes with every update. We forked and extensively modified these projects to resolve conflicts, implement missing features, and fix blocking bugs. Many modifications were contributed back upstream, but each library update frequently broke the entire system, requiring substantial effort to restore compatibility. This experience highlights an inherent tension in cutting-edge research where leveraging the latest capabilities requires accepting integration burden that can exceed core research implementation effort.

6.4.2 Reward Misspecification

Initial experiments with auxiliary rewards revealed subtle failure modes in multi-objective training. We introduced a file-matching reward to provide intermediate learning signal, hypothesizing it would help the agent learn to identify relevant files before optimizing patch quality. However, because the primary diff similarity reward remained sparse while the file-matching reward provided frequent feedback, the model optimized aggressively for the auxiliary objective to the detriment of actual patch similarity.

Evaluating successive training checkpoints on SWE-Bench-Verified exposed this misspecification clearly. Downstream performance improved initially but noticeably worsened after a certain point in training, even as file-matching accuracy continued increasing. The agent learned to modify the correct files consistently while producing increasingly poor patches, a textbook case of reward hacking where optimizing a proxy metric degraded the true objective. This experience reinforced the value of sparse, well-aligned rewards over dense but misspecified shaping signals.

6.5 Broader Reflections on RL

The application of RL to real-world tasks like coding represents a fundamental shift in what reinforcement learning can accomplish. AlphaZero's superhuman game performance demonstrated the power of learning through interaction, yet simultaneously exposed a critical limitation: games provide reward signals for every legal move, while real-world tasks require deep conceptual understanding just to participate meaningfully. Before LLMs, applying RL to tasks like code repair was futile not because the algorithms were inadequate, but because models lacked the prerequisite knowledge to attempt such problems coherently. An RL agent cannot learn from experience if it cannot generate experiences worth learning from; there is no gradient to climb when every action produces meaningless output.

LLMs solved this foundational problem by providing the conceptual substrate necessary for coherent interaction with complex environments. When an LLM based RL agent attempts to fix a bug, it can execute valid commands, parse error messages, and propose syntactically reasonable patches. These capabilities, acquired through massive-scale pretraining, give the agent that crucial first rung on the ladder, enabling the generation of rich learning signals that RL algorithms can exploit. For the first time, we can apply decades of reinforcement learning research to problems that matter, because LLMs have bridged the gap between random exploration and meaningful interaction. This thesis demonstrates that potential in practice: online RL training for coding agents is feasible within academic compute constraints and yields measurable improvements in scaffold adaptation. The convergence of LLMs and RL enables training approaches previously confined to simple domains with well-defined action spaces to extend to complex reasoning tasks like repository-level debugging. By demonstrating that debugging behaviors emerge from simple tools combined with RL training, this thesis provides evidence that experiential learning complements supervised approaches for interactive agent development.

The infrastructure and methodologies developed here offer a foundation for future investigations into how agents acquire complex skills through environmental interaction, extending beyond coding to domains where LLMs provide sufficient baseline capabilities for meaningful participation. Perhaps most importantly, this work demonstrates that sophisticated AI research infrastructure can be developed within academic settings through careful engineering and open-source collaboration. By providing accessible implementations of online RL training systems, we enable broader investigation of experiential learning approaches that might otherwise remain confined to well-resourced industry laboratories.

The core principles validated here, including execution-free rewards enabling language-agnostic training, live weight synchronization enabling efficient multi-turn agent training, and systematic optimization making online RL tractable within academic budgets, suggest productive directions for future research. Whether these approaches scale to match frontier capabilities or require fundamental architectural advances remains an open question, but the foundation established here demonstrates that open, reproducible research in experiential learning for coding agents is both viable and valuable. The questions worth asking are now more clearly defined, and the tools to investigate them are available to the broader research community.

Bibliography

[1] Anthropic. *Claude 4: Building a New Era of AI Assistant Capabilities*. https://www-cdn.anthropic.com/07b2a3f9902ee19fe39a36ca638e5ae987bc64dd.pdf. Technical report detailing Claude 4's performance on SWE-Bench-Verified and other benchmarks. 2025.

[2] Anthropic PBC. *Claude Code: Overview and Docs*. Product documentation. URL: https://docs.anthropic.com/en/docs/claude-code/overview (visited on 08/23/2025).

[3] Chen, Mark, Tworek, Jerry, Jun, Heewoo, Yuan, Qiming, Oliveira Pinto, Henrique Ponde de, Kaplan, Jared, Edwards, Harri, Burda, Yuri, Joseph, Nicholas, Brockman, Greg, Ray, Alex, Puri, Raul, Krueger, Gretchen, Petrov, Michael, Khlaaf, Heidy, Sastry, Girish, Mishkin, Pamela, Chan, Brooke, Gray, Scott, Ryder, Nick, Pavlov, Mikhail, Power, Alethea, Kaiser, Lukasz, Bavarian, Mohammad, Winter, Clemens, Tillet, Philippe, Such, Felipe Petroski, Cummings, Dave, Plappert, Matthias, Chantzis, Fotios, Barnes, Elizabeth, Herbert-Voss, Ariel, Guss, William Hebgen, Nichol, Alex, Paino, Alex, Tezak, Nikolas, Tang, Jie, Babuschkin, Igor, Balaji, Suchir, Jain, Shantanu, Saunders, William, Hesse, Christopher, Carr, Andrew N., Leike, Jan, Achiam, Josh, Misra, Vedant, Morikawa, Evan, Radford, Alec, Knight, Matthew, Brundage, Miles, Murati, Mira, Mayer, Katie, Welinder, Peter, McGrew, Bob, Amodei, Dario, McCandlish, Sam, Sutskever, Ilya, and Zaremba, Wojciech. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG]. URL: https://arxiv.org/abs/2107.03374.

[4] Google. *Gemini CLI*. Version v0.3.0-nightly.20250823.1a89d185. GitHub release tag; use a stable tag if citing a non-nightly build. Aug. 23, 2025. URL: https://github.com/google-gemini/gemini-cli (visited on 08/23/2025).

[5] Hu, Edward J, Shen, Yelong, Wallis, Phillip, Allen-Zhu, Zeyuan, Li, Yuanzhi, Wang, Shean, Wang, Lu, and Chen, Weizhu. “LoRA: Low-Rank Adaptation of Large Language Models.” In: *arXiv preprint arXiv:2106.09685* (2021).

[6] Jimenez, Carlos E., Yang, John, Wettig, Alexander, Yao, Shunyu, Pei, Kexin, Press, Ofir, and Narasimhan, Karthik. *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?* 2024. arXiv: 2310.06770 [cs.CL]. URL: https://arxiv.org/abs/2310.06770.

[7] Just, René, Jalali, Darioush, and Ernst, Michael D. “Defects4J: a database of existing faults to enable controlled testing studies for Java programs.” In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. San Jose, CA, USA: Association for Computing Machinery, 2014, pp. 437–440. ISBN: 9781450326452. DOI: 10.1145/2610384.2628055. URL: https://doi.org/10.1145/2610384.2628055.

[8] Kim, Dongsun, Nam, Jaechang, Song, Jaewoo, and Kim, Sunghun. “Automatic patch generation learned from human-written patches.” In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 802–811. DOI: 10.1109/ICSE.2013.6606626.

[9] Le, Hung, Wang, Yue, Gotmare, Akhilesh Deepak, Savarese, Silvio, and Hoi, Steven C. H. *CodeRL: Mastering Code Generation through Pretrained Models and Deep Reinforcement Learning*. 2022. arXiv: 2207.01780 [cs.LG]. URL: https://arxiv.org/abs/2207.01780.

[10] Le Goues, Claire, Nguyen, ThanhVu, Forrest, Stephanie, and Weimer, Westley. “GenProg: A Generic Method for Automatic Software Repair.” In: *IEEE Transactions on Software Engineering* 38.1 (2012), pp. 54–72. DOI: 10.1109/TSE.2011.104.

[11] Lin, Derrick, Koppel, James, Chen, Angela, and Solar-Lezama, Armando. “QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge.” In: *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. SPLASH Companion 2017. Vancouver, BC, Canada: Association for Computing Machinery, 2017, pp. 55–56. ISBN: 9781450355148. DOI: 10.1145/3135932.3135941. URL: https://doi.org/10.1145/3135932.3135941.

[12] Liu, Zichen, Chen, Changyu, Li, Wenjun, Qi, Penghui, Pang, Tianyu, Du, Chao, Lee, Wee Sun, and Lin, Min. *Understanding R1-Zero-Like Training: A Critical Perspective*. 2025. arXiv: 2503.20783 [cs.LG]. URL: https://arxiv.org/abs/2503.20783.

[13] Long, Fan and Rinard, Martin. "Automatic patch generation by learning correct code." In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. St. Petersburg, FL, USA: Association for Computing Machinery, 2016, pp. 298–312. ISBN: 9781450335492. DOI: 10.1145/2837614.2837617. URL: https://doi.org/10.1145/2837614.2837617.

[14] Luo, Michael, Jain, Naman, Singh, Jaskirat, Tan, Sijun, Patel, Ameen, Wu, Qingyang, Ariyak, Alpay, Cai, Colin, Venkat, Tarun, Zhu, Shang, Athiwaratkun, Ben, Roongta, Manan, Zhang, Ce, Li, Li Erran, Popa, Raluca Ada, Sen, Koushik, and Stoica, Ion. *DeepSWE: Training a State-of-the-Art Coding Agent from Scratch by Scaling RL*. https://pretty-radio-b75.notion.site/DeepSWE-Training-a-Fully-Open-sourced-State-of-the-Art-Coding-Agent-by-Scaling-RL-22281902c1468193aabbe9a8c59bbe33. Notion Blog. 2025.

[15] Nguyen, Hoang Duong Thien, Qi, Dawei, Roychoudhury, Abhik, and Chandra, Satish. "SemFix: Program repair via semantic analysis." In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 772–781. DOI: 10.1109/ICSE.2013.6606623.

[16] OpenAI. *Codex CLI*. Version rust-v0.23.0. GitHub release tag rust-v0.23.0. Aug. 20, 2025. URL: https://github.com/openai/codex (visited on 08/23/2025).

[17] OpenAI. "Introducing SWE-bench Verified." In: (2024). Updated 2025-02-24. Accessed 2025-08-23.

[18] Pan, Jiayi, Wang, Xingyao, Neubig, Graham, Jaitly, Navdeep, Ji, Heng, Suhr, Alane, and Zhang, Yizhe. *Training Software Engineering Agents and Verifiers with SWE-Gym*. 2025. arXiv: 2412.21139 [cs.SE]. URL: https://arxiv.org/abs/2412.21139.

[19] Qwen Team. *Qwen Code*. Command-line agent harness; GitHub repository. URL: https://github.com/QwenLM/qwen-code (visited on 08/23/2025).

[20] Schulman, John, Wolski, Filip, Dhariwal, Prafulla, Radford, Alec, and Klimov, Oleg. *Proximal Policy Optimization Algorithms*. 2017. arXiv: 1707.06347 [cs.LG]. URL: https://arxiv.org/abs/1707.06347.

[21] Shao, Zhihong, Wang, Peiyi, Zhu, Qihao, Xu, Runxin, Song, Junxiao, Bi, Xiao, Zhang, Haowei, Zhang, Mingchuan, Li, Y. K., Wu, Y., and Guo, Daya. *DeepSeek-Math: Pushing the Limits of Mathematical Reasoning in Open Language Models*. 2024. arXiv: 2402.03300 [cs.CL]. URL: https://arxiv.org/abs/2402.03300.

[22] Silva, André, Fang, Sen, and Monperrus, Martin. "RepairLLaMA: Efficient Representations and Fine-Tuned Adapters for Program Repair." In: *arXiv preprint arXiv:2312.15698* (2023).

[23] Silva, André, Saavedra, Nuno, and Monperrus, Martin. "GitBug-Java: A Reproducible Benchmark of Recent Java Bugs." In: *Proceedings of the 21st International Conference on Mining Software Repositories*. MSR '24. ACM, Apr. 2024, pp. 118–122. DOI: 10.1145/3643991.3644884. URL: http://dx.doi.org/10.1145/3643991.3644884.

[24] Silver, David, Huang, Aja, Maddison, Chris J., Guez, Arthur, Sifre, Laurent, Driessche, George van den, Schrittwieser, Julian, Antonoglou, Ioannis, Panneershelvam, Veda, Lanctot, Marc, Dieleman, Sander, Grewe, Dominik, Nham, John, Kalchbrenner, Nal, Sutskever, Ilya, Lillicrap, Timothy, Leach, Madeleine, Kavukcuoglu, Koray, Graepel, Thore, and Hassabis, Demis. "Mastering the game of Go with deep neural networks and tree search." In: *Nature* 529.7587 (2016), pp. 484–489. DOI: 10.1038/nature16961.

[25] Team, 5 et al. *GLM-4.5: Agentic, Reasoning, and Coding (ARC) Foundation Models*. 2025. arXiv: 2508.06471 [cs.CL]. URL: https://arxiv.org/abs/2508.06471.

[26] Team, Kimi et al. *Kimi K2: Open Agentic Intelligence*. 2025. arXiv: 2507.20534 [cs.LG]. URL: https://arxiv.org/abs/2507.20534.

[27] Team, Qwen. *Qwen3-Coder: Agentic Coding in the World*. https://qwenlm.github.io/blog/qwen3-coder/. Blog post introducing Qwen3-Coder capabilities. 2025.

[28] Wang, Xingyao, Chen, Boxuan, Li, Yufan, Zhang, Bowen, Liu, Zhengyang, Gao, Cheng, Huang, Yue, Li, Guanzhi, Zhang, Nan, Xu, Mengnan, et al. *OpenHands: An Open Platform for AI Software Developers as Generalist Agents*. 2024.

[29] Wei, Yuxiang, Duchenne, Olivier, Copet, Jade, Carbonneaux, Quentin, Zhang, Lingming, Fried, Daniel, Synnaeve, Gabriel, Singh, Rishabh, and Wang, Sida I. *SWE-RL: Advancing LLM Reasoning via Reinforcement Learning on Open Software Evolution*. 2025. arXiv: 2502.18449 [cs.SE]. URL: https://arxiv.org/abs/2502.18449.

[30] Xia, Chunqiu Steven, Deng, Yinlin, Dunn, Soren, and Zhang, Lingming. *Agentless: Demystifying LLM-based Software Engineering Agents*. 2024. arXiv: 2407.01489 [cs.SE]. URL: https://arxiv.org/abs/2407.01489.

[31] Yang, John, Jimenez, Carlos E, Wettig, Alexander, Lieret, Kilian, Yao, Shunyu, Narasimhan, Karthik R, and Press, Ofir. *mini-swe-agent: The 100 line AI agent that solves GitHub issues*. https://github.com/SWE-agent/mini-swe-agent. GitHub repository. 2025.

[32] Yang, John, Jimenez, Carlos E., Wettig, Alexander, Lieret, Kilian, Yao, Shunyu, Narasimhan, Karthik, and Press, Ofir. *SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering*. 2024. arXiv: 2405.15793 [cs.SE]. URL: https://arxiv.org/abs/2405.15793.

[33] Yang, John, Lieret, Kilian, Jimenez, Carlos E., Wettig, Alexander, Khandpur, Kabir, Zhang, Yanzhe, Hui, Binyuan, Press, Ofir, Schmidt, Ludwig, and Yang, Diyi. *SWE-smith: Scaling Data for Software Engineering Agents*. 2025. arXiv: 2504.21798 [cs.SE]. URL: https://arxiv.org/abs/2504.21798.

[34] Yao, Shunyu, Shinn, Noah, Razavi, Pedram, and Narasimhan, Karthik. *τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains*. 2024. arXiv: 2406.12045 [cs.AI]. URL: https://arxiv.org/abs/2406.12045.

[35] Yu, Qiying, Zhang, Zheng, Zhu, Ruofei, Yuan, Yufeng, Zuo, Xiaochen, Yue, Yu, Dai, Weinan, Fan, Tiantian, Liu, Gaohong, Liu, Lingjun, Liu, Xin, Lin, Haibin, Lin, Zhiqi, Ma, Bole, Sheng, Guangming, Tong, Yuxuan, Zhang, Chi, Zhang, Mofan, Zhang, Wang, Zhu, Hang, Zhu, Jinhua, Chen, Jiaze, Chen, Jiangjie, Wang, Chengyi, Yu, Hongli, Song, Yuxuan, Wei, Xiangpeng, Zhou, Hao, Liu, Jingjing, Ma, Wei-Ying, Zhang, Ya-Qin, Yan, Lin, Qiao, Mu, Wu, Yonghui, and Wang, Mingxuan. *DAPO: An Open-Source LLM Reinforcement Learning*

System at Scale. 2025. arXiv: 2503.14476 [cs.LG]. URL: https://arxiv.org/abs/2503.14476.

[36] Zheng, Chujie, Liu, Shixuan, Li, Mingze, Chen, Xiong-Hui, Yu, Bowen, Gao, Chang, Dang, Kai, Liu, Yuqiong, Men, Rui, Yang, An, Zhou, Jingren, and Lin, Junyang. *Group Sequence Policy Optimization*. 2025. arXiv: 2507.18071 [cs.LG]. URL: https://arxiv.org/abs/2507.18071.

Appendix - Contents

Appendix A

Limited Size and Model Family Ablation

The choice of Qwen3-14B as the primary model for this thesis emerged from systematic exploration of multiple model families and sizes, balancing computational constraints with task performance. At the time of experimentation, Qwen3 was widely regarded as the strongest open-weight coding model, particularly for tool-augmented tasks. Qwen3-14B offered an optimal fit for available resources: Qwen3-8B proved too large for two GPUs yet too small to justify three, while the 32B variant though promising, imposed prohibitive computational demands for exploratory training runs.

Beyond capacity considerations, the combination of tool-calling capability and sufficient reasoning ability proved critical for agentic RL training. Llama3.1-8B, while competent at formatting tool calls, exhibited near-zero rewards throughout preliminary training, suggesting it lacked the problem-solving intelligence necessary to use tools effectively for repository navigation and bug localization. Gemma3's tool-calling interface did not adapt well to the structured command sequences required for repository navigation and code editing. Ministral substantially underperformed compared to more recent releases. GLM4-9B showed promise but suffered from inadequate support in the Transformers and vLLM ecosystems at the time of experimentation. Similarly, Llama3.1-Nemotron (Nvidia) demonstrated potential but would have required extensive vLLM refactoring to accommodate its custom tool-calling parser.

These constraints narrowed the viable candidates to the Qwen3 family, where robust tool-calling support, strong baseline coding performance, and flexible scaling across 8B, 14B, and 32B enabled systematic capacity analysis within consistent infrastructure.

Figures A.0.1 and A.0.2 compare training trajectories across model families. Llama3.1-8B plateaus near zero reward with minimal tool success, motivating its exclusion from main results. Qwen3-32B reaches 0.20 reward before early termination at step 135 due to compute constraints. Qwen3-14B and 8B exhibit comparable performance trajectories.

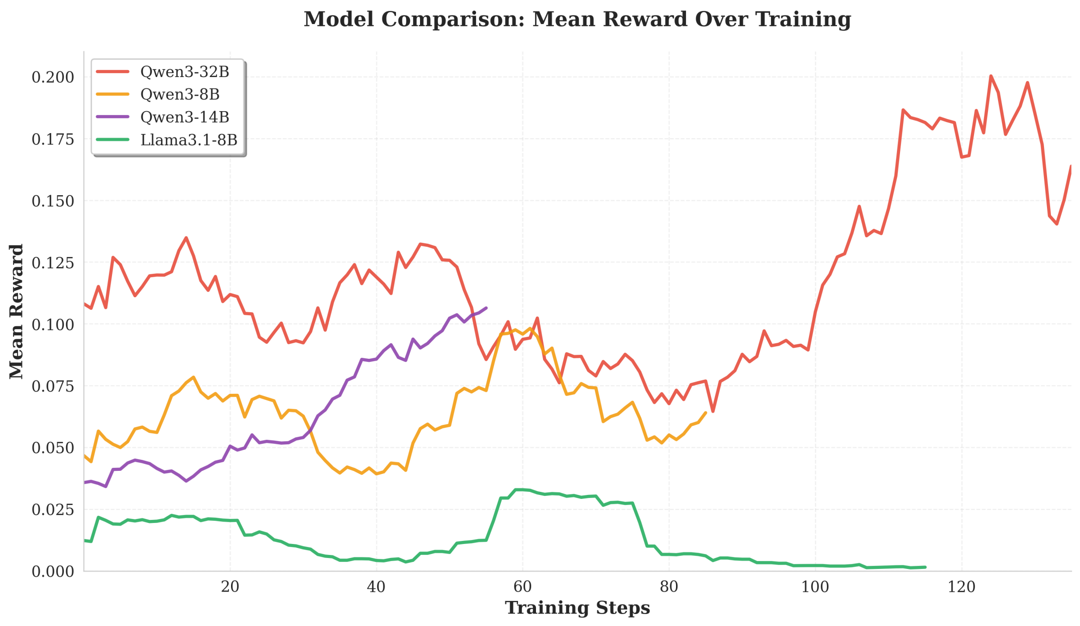

Figure A.0.1: Mean reward progression across model families and sizes, plotted with rolling window average.

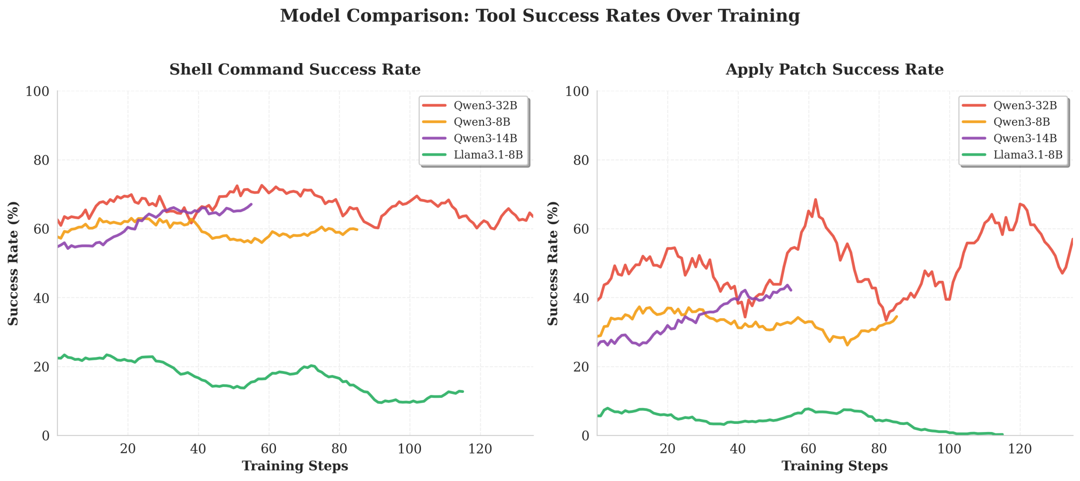

Figure A.0.2: Tool success rates across models. Left: shell commands. Right: apply_patch.

Appendix B

Nano Agent Benchmark

During Nano development, we established this benchmark to track performance improvements across agent versions and design iterations, initially validating changes against GPT-4.1-mini. Without systematic evaluation, determining whether alternative tooling strategies or prompt variations improved performance would have been infeasible given the stochastic nature of agent behavior and the lack of deterministic success metrics during development.

The benchmark operates identically to the inference component of our training system: Nano serves as the agent scaffold while hosted language models execute terminal commands to repair buggy GitHub repositories. The evaluation set consists of a 10-task subset drawn from SWE-Bench-Lite, with each reported metric representing the average of 1 to 8 runs per model, with run counts determined pragmatically by API cost constraints. Solutions are measured against ground truth using code similarity (patch matching with the actual bug fix, serving as the primary ranking metric) and test similarity (alignment of test changes with ground truth test updates).

Populating the benchmark with frontier model performance provides a useful validation: the resulting rankings align closely with general community consensus on agentic coding performance, with Claude Sonnet 4 achieving the highest code similarity scores (0.394). Models exhibit substantial variation in token efficiency, with DeepSeek Chat consuming only 3,297 tokens while Claude Sonnet 4 approaches the 16,384-token limit, reflecting different exploration strategies and verbosity patterns. Most models achieve low test similarity scores, suggesting that generating appropriate test modifications alongside code fixes remains challenging even for frontier systems.

Interestingly, frontier models displayed a tendency to implement test cases despite no such requirement in the task prompt, likely reflecting training practices carefully cultivated in closed-source research labs.

Table B.0.1: Nano Agent Benchmark Performance on subset of SWE-bench Lite. Error bars on these results are substantial given the small evaluation set (10 tasks) and variable run counts (1 to 8 per model).

Rank	Version	Model	Code Sim	Test Sim	Tokens	Tools
1	v3.2.0	claude-sonnet-4-20250514	0.394	0.188	14,746 / 16,384	41.5 / 100
2	v3.2.0	gpt-4.1	0.387	0.092	9,777 / 16,384	35.7 / 100
3	v4.0.2	deepseek-chat-v3.1	0.383	0.078	11,762 / 16,384	41.9 / 100
4	v4.0.1	kimi-k2	0.382	0.009	5,508 / 16,384	19.7 / 100
5	v4.0.1	qwen3-coder	0.374	0.042	6,979 / 16,384	26.5 / 100
6	v3.2.0	gemini-2.5-pro-preview	0.370	0.034	6,008 / 16,384	13.6 / 100
7	v3.3.0	gemini-2.5-flash	0.363	0.022	4,337 / 16,384	13.2 / 100
8	v3.2.0	gpt-4.1-mini	0.350	0.017	7,403 / 16,384	29.7 / 100
9	v3.2.0	deepseek-chat	0.336	0.011	3,297 / 16,384	7.5 / 100
10	v4.0.1	glm-4.5	0.323	0.107	12,477 / 16,384	28.7 / 100
11	v3.2.0	qwen-2.5-72b-instruct	0.272	0.000	5,873 / 16,384	35.1 / 100
12	v3.2.0	qwen3-32b	0.255	0.000	5,281 / 16,384	28.3 / 100
13	v3.2.0	llama-4-maverick	0.255	0.000	4,647 / 16,384	10.4 / 100
14	v3.2.0	qwen3-8b	0.190	0.000	8,704 / 16,384	56.5 / 100
15	v3.2.0	gpt-4.1-nano	0.188	0.000	8,536 / 16,384	33.1 / 100
16	v3.2.0	qwen3-14b	0.176	0.000	10,800 / 16,384	82.6 / 100
17	v3.2.0	devstral-small	0.092	0.000	14,603 / 16,384	13.0 / 100

Appendix C

Reproducibility and Contribution

All training recipes, infrastructure code, and evaluation pipelines are released as open-source software with accessible documentation and straightforward setup procedures. The main repository at https://github.com/ASSERT-KTH/CodeRepairRL contains complete training scripts, SLURM job templates, and episode orchestration logic. The thesis repository at https://github.com/BjarniHaukur/CodeRepairRL-Paper contains the LaTeX source for this document alongside all analysis and plotting scripts, which use the Weights & Biases API to access logged experimental data to generate figures.

The Nano agent implementation is available at https://github.com/ASSERT-KTH/nano-agent, providing the minimal terminal-based scaffold used throughout all experiments. Our TRL fork at https://github.com/ASSERT-KTH/trl/tree/main extends the upstream library with modifications for multi-turn agent trajectories, GSPO support, and asynchronous live weight synchronization with vLLM extensions and NCCL channels. The GSPO kernel implementation contributed to Liger-Kernel is documented at https://github.com/linkedin/Liger-Kernel/pull/845.

Appendix D

Use of AI Assistance

AI language models, primarily Anthropic's Claude Sonnet (3.5, 4 and 4.5), OpenAI's o3, and GPT-5, were used extensively throughout this research as coding assistants, writing aids, and research tools. AI assistance included writing and debugging implementation code, improving prose clarity and academic style, organizing research notes, researching background literature and technical documentation, and formatting LaTeX markup. All research directions, experimental designs, and scientific interpretations originated with me.