DEGREE PROJECT IN TECHNOLOGY,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2025

CodeRepairRL

KTH Thesis Report

Bjarni Haukur

KTH ROYAL INSTITUTE OF TECHNOLOGY
ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Authors

Bjarni <email@domain.com> and Author Name
Information and Communication Technology
KTH Royal Institute of Technology

Place for Project

Stockholm, Sweden
Some place

Examiner

Martin Place
KTH Royal Institute of Technology

Supervisor

The Supervisor

Place

KTH Royal Institute of Technology

Abstract

This is a template for writing thesis reports for the ICT school at KTH. I do not own any of the images provided in the template and this can only be used to submit thesis work for KTH.

The report needs to be compiled using XeLaTeX as different fonts are needed for the project to look like the original report. You might have to change this manually in overleaf.

This template was created by Hannes Rabo <hannes.rabo@gmail.com or hrabo@kth.se> from the template provided by KTH. You can send me an email if you need help in making it work for you.

Write an abstract. Introduce the subject area for the project and describe the problems that are solved and described in the thesis. Present how the problems have been solved, methods used and present results for the project. Use probably one sentence for each chapter in the final report.

The presentation of the results should be the main part of the abstract. Use about ½ A4-page. English abstract

Keywords

Template, Thesis, Keywords ...

Abstract

Svenskt abstract Svensk version av abstract – samma titel på svenska som på engelska.

Skriv samma abstract på svenska. Introducera ämnet för projektet och beskriv problemen som löses i materialet. Presentera

Nyckelord

Kandidat examensarbete, ...

Acknowledgements

Write a short acknowledgements. Don’t forget to give some credit to the examiner and supervisor.

Acronyms

Contents

Chapter 1

Introduction

1.1 Background

Automated code repair has emerged as one of the most challenging problems in software engineering. Despite significant advances in large language models (LLMs), even state-of-the-art systems solve under 20% of real bug-fix tasks in rigorous benchmarks like SWE-Bench. Traditional approaches to training code repair models rely on passive learning from static datasets of bug-fix pairs, where models observe input-output examples without ever experiencing the iterative, exploratory process that human developers use when debugging.

The evolution of AI capabilities provides crucial context for understanding why reinforcement learning (RL) has only recently become applicable to complex real-world tasks like code repair. AlphaZero's superhuman game performance represented a significant milestone in AI development, yet it simultaneously exposed a fundamental limitation: games provide reward signals for every legal move, while real-world tasks require deep conceptual understanding merely to participate meaningfully. Before the advent of LLMs, applying RL to tasks like coding remained futile—models could not even attempt such problems coherently. The limitation was not algorithmic; rather, these systems lacked any gradient to climb.

LLMs fundamentally transformed this landscape by providing the prerequisite knowledge to attempt real-world tasks coherently. When an RL agent attempts to fix a bug, it can now execute valid commands, interpret error messages, and generate the rich learning signals that RL algorithms require. For the first time, decades of RL research can be applied to problems of practical significance, as LLMs have provided that crucial first rung on the ladder of capability.

This limitation becomes particularly apparent when considering how humans actually fix bugs. Developers don't simply look at broken code and immediately produce a patch. Instead, they navigate through codebases, examine multiple files, run tests, execute commands, and iteratively refine their understanding before implementing a solution. This active, exploratory process is fundamentally different from the passive pattern matching that current LLMs perform.

Recent work has begun exploring agent-based approaches where LLMs are given tools to interact with code repositories. However, these systems are typically trained using traditional supervised learning on static datasets, then deployed as agents at inference time. This disconnect between training and deployment environments may limit their effectiveness.

1.2 Problem

The core problem addressed in this thesis is that current approaches to automated code repair suffer from a fundamental mismatch between how models are trained and how they are deployed. Models learn from static bug-fix examples but are expected to perform dynamic, multi-step debugging in interactive environments.

Specifically, we identify three key limitations in existing approaches:

First, **passive learning limits exploration capabilities**. Models trained on static datasets never learn to actively explore codebases, navigate file structures, or iteratively gather information needed for complex bug fixes.

Second, **single-step generation doesn't match debugging workflows**. Real debugging involves multiple iterations of hypothesis formation, testing, and refinement, but current models are trained to produce patches in a single forward pass.

Third, **scaffold complexity effects are poorly understood**. When LLMs are deployed as coding agents, they rely on various tools and interfaces (scaffolds) to interact with code. It's unclear whether simple, minimal scaffolds or complex, feature-rich ones lead to better learning outcomes when integrated into training.

How can we train LLMs to perform automated code repair through active, agent-like interaction with codebases, and what role does scaffold complexity play in this learning process?

1.3 Purpose

The purpose of this thesis is to investigate a novel training paradigm called ”agent-in-the-loop reinforcement learning” for automated code repair. Rather than training models on static datasets, we embed coding agents directly into the reinforcement learning training loop, allowing models to learn through active interaction with real codebases.

This work aims to demonstrate that LLMs can learn more effective debugging strategies when they experience the full process of navigating, exploring, and iteratively fixing bugs, rather than simply observing input-output pairs. We specifically examine how different levels of scaffold complexity - from minimalist terminal-based interfaces to feature-rich development environments - affect learning outcomes.

1.4 Goal

The primary goal of this research is to develop and evaluate an agent-in-the-loop reinforcement learning framework for training coding agents to perform automated program repair.

Specific deliverables include:

- A novel training pipeline integrating coding agents directly into GRPO (Group Relative Policy Optimization) reinforcement learning loops
- Implementation of two contrasting agent scaffolds: a minimalist ”nano-agent” using only basic terminal commands, and a heavyweight scaffold with rich contextual features
- Experimental evaluation on Python bug-fixing tasks using SWE-Gym and SWE-Bench-Verified datasets
- Analysis of whether reinforcement learning produces monotonic improvement in coding agent performance - a result that would represent significant progress

in open-source automated debugging research

1.5 Research Questions

This thesis is guided by three primary research questions:

RQ1: Does integrating an agent scaffold into the reinforcement learning training loop significantly improve automated code repair performance compared to non-RL-trained models?

This foundational question investigates whether the agent-in-the-loop RL approach produces measurable improvements over baseline model performance. We hypothesize that learning through active interaction will yield better debugging capabilities than passive learning alone.

RQ2: Does a minimalist coding scaffold yield better performance in RL-based code repair than a heavyweight scaffold with extensive engineered features?

Inspired by the 'bitter lesson' in AI research, this question examines whether simple, general-purpose tools enable better learning outcomes than hand-engineered, feature-rich environments. We compare a minimalist nano-agent (using only shell commands and basic file operations) against heavyweight scaffolds with repository mapping, context summarization, and guided reasoning.

RQ3: Do the performance gains from scaffold-in-the-loop RL training generalize beyond the training environment to other programming languages and general code generation tasks?

This question probes whether the approach produces fundamental improvements in reasoning and problem-solving abilities, or merely optimizes performance within the specific training scaffold and language. We evaluate generalization to Java bug-fixing and general code generation benchmarks.

1.6 Methodology

This research employs an experimental methodology combining reinforcement learning techniques with agent-based software engineering environments. The core approach involves a two-stage training pipeline: supervised fine-tuning (SFT) on high-quality code repair demonstrations, followed by Group Relative Policy Optimization (GRPO) reinforcement learning where agents actively interact with codebases to fix bugs.

The experimental design compares multiple agent scaffold configurations across standardized benchmarks, using both quantitative metrics (patch similarity, task success rates) and qualitative analysis of learning trajectories. We employ controlled experiments to isolate the effects of scaffold complexity while maintaining consistent model architectures and training procedures.

Data collection involves automated interaction logs from agent-codebase interactions, reward signals based on patch quality, and evaluation results across multiple programming languages and task types.

1.7 Delimitations

This research focuses specifically on automated bug fixing rather than broader code generation tasks, though we include limited evaluation on general programming benchmarks for generalization assessment. The primary evaluation environment is Python codebases, with Java evaluation for cross-language generalization.

The scope is limited to repository-level bug fixes rather than system-level or infrastructure issues. We do not address bugs requiring external dependencies, network interactions, or complex runtime environments beyond what can be simulated in containerized settings.

While the approach is designed to be model-agnostic, experiments focus on Qwen3 family models due to computational constraints. The reinforcement learning approach requires significant computational resources, limiting the scale of hyperparameter exploration.

1.8 Outline

Chapter 2 provides comprehensive background on reinforcement learning for language models, automated code repair, and agent-based programming environments, situating this work within existing literature.

Chapter 3 presents the detailed methodology, including the agent-in-the-loop training pipeline, scaffold implementations, and experimental design. Chapter 4 describes the technical implementation of the training infrastructure and agent frameworks. Chapter 5 presents experimental results and analysis, focusing on the core research questions and learning trajectory analysis. Chapter 6 concludes with discussion of findings, limitations, and future research directions.

Chapter 2

Background and Related Work

This chapter provides the theoretical foundation for agent-in-the-loop reinforcement learning applied to automated code repair. We review key concepts in reinforcement learning for language models, automated program repair, and agent-based software engineering approaches.

2.1 Automated Code Repair with Language Models

Large language models have transformed automated program repair, moving from rule-based and search-based approaches to neural methods capable of understanding complex code patterns and generating sophisticated patches.

TODO: Cover evolution from early APR systems (GenProg, etc.) to neural approaches. Discuss key datasets like Defects4J, CodeXGLUE. Review state-of-the-art LLM performance on benchmarks like SWE-Bench (currently 20% success rates). Highlight limitations: single-step generation, lack of iterative refinement, poor handling of multi-file changes. Cite recent work like RepairLLaMA, CodeT5 variants.

Current approaches primarily rely on supervised fine-tuning on bug-fix datasets, where models learn to map broken code snippets to corrected versions. However, this paradigm has fundamental limitations when applied to real-world debugging scenarios.

TODO: Discuss the mismatch between training (static input-output pairs) and deployment (interactive debugging). Explain why simple sequence-to-sequence models struggle with repository-level understanding and complex multi-step reasoning required for realistic bug fixes.

2.2 Reinforcement Learning for Language Models

Reinforcement learning has emerged as a powerful technique for improving language model performance beyond what supervised learning alone can achieve, particularly for tasks requiring sequential decision-making and optimization of complex objectives. Unlike supervised learning, which relies on fixed input-output pairs, RL enables models to learn through interaction with dynamic environments, receiving rewards based on the quality of their generated outputs.

2.2.1 Policy Gradient Foundations

The foundation of RL for language models lies in treating text generation as a Markov Decision Process (MDP). In this formulation, the language model serves as a policy $\pi_\theta(a_t|s_t)$ that selects actions (tokens) a_t given states (context sequences) s_t, parameterized by model weights θ.

The objective is to maximize the expected cumulative reward:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)] \tag{2.1}$$

where $\tau = (s_0, a_0, s_1, a_1, \ldots, s_T, a_T)$ represents a trajectory (complete sequence generation) and $R(\tau)$ is the total reward for that trajectory.

The policy gradient theorem provides the foundation for optimization:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)] \tag{2.2}$$

However, this basic REINFORCE estimator suffers from high variance, making training unstable and sample-inefficient. This limitation becomes particularly problematic for language models, where sequence lengths can be substantial and reward signals are often sparse.

2.2.2 Proximal Policy Optimization (PPO)

Proximal Policy Optimization has emerged as the dominant algorithm for fine-tuning large language models due to its ability to stabilize training while maintaining sample efficiency. PPO addresses the fundamental challenge of policy optimization: making meaningful progress without taking overly large steps that destabilize learning.

The Clipping Mechanism

PPO introduces a clipped objective function that prevents destructive policy updates:

$$L^{CLIP}(\theta) = \mathbb{E}_t \left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right] \tag{2.3}$$

where:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{2.4}$$

$$\hat{A}_t = R_t - V(s_t) \tag{2.5}$$

The ratio $r_t(\theta)$ measures how much the current policy differs from the previous policy, while $\hat{A}_t$ represents the advantage estimate computed using a value function $V(s_t)$. The clipping parameter ϵ (typically 0.2) constrains policy updates to prevent catastrophic changes.

Value Function Training

PPO employs a separate value network $V_\phi(s)$ trained to predict expected returns, enabling more accurate advantage estimation:

$$L^{VF}(\phi) = \mathbb{E}_t\left[(V_\phi(s_t) - R_t)^2\right] \tag{2.6}$$

The complete PPO objective combines policy and value losses:

$$L(\theta, \phi) = L^{CLIP}(\theta) - c_1 L^{VF}(\phi) + c_2 S[\pi_\theta](s_t) \tag{2.7}$$

where $S[\pi_\theta]$ is an entropy bonus encouraging exploration, and c_1, c_2 are weighting coefficients.

Success in Language Model Fine-tuning

PPO's success in language model applications, particularly in Reinforcement Learning from Human Feedback (RLHF), stems from several key properties:

Stable Learning: The clipping mechanism prevents the policy from changing too rapidly, which is crucial when fine-tuning large pre-trained models where dramatic changes can destroy learned representations.

Sample Efficiency: By reusing data for multiple gradient steps and employing importance sampling correction, PPO achieves better sample efficiency than simpler policy gradient methods.

Scalability: PPO's architecture separates policy and value training, enabling distributed training across multiple GPUs with different computational loads for each component.

However, PPO also introduces significant computational overhead through the separate value network training and the need for multiple gradient updates per batch of experience.

2.2.3 Group Relative Policy Optimization (GRPO)

Group Relative Policy Optimization is fundamentally PPO with a crucial simplification: instead of training a separate value network to estimate advantages, GRPO computes advantages directly from relative performance within sampled action groups. This elegant modification preserves PPO's theoretical guarantees while dramatically reducing computational overhead.

The Core Simplification

The key insight behind GRPO is that for each state s, rather than estimating $V(s)$ with a separate network, we can sample multiple actions $a_1, \ldots, a_G$ from the current policy π_{θ_t} and use their reward distribution to compute relative advantages.

For a given state s, GRPO samples G actions from the policy and computes the group-relative advantage as:

$$A^{\pi_{\theta_t}}(s, a_j) = \frac{r(s, a_j) - \mu}{\sigma} \tag{2.8}$$

where μ and σ are the mean and standard deviation of the rewards $r(s, a_1), \ldots, r(s, a_G)$.

This is simply the standard score (z-score) of the rewards, providing a normalized measure of relative performance.

Mathematically, this can be expressed as:

$$\mu = \frac{1}{G}\sum_{i=1}^{G} r(s, a_i) \tag{2.9}$$

$$\sigma = \sqrt{\frac{1}{G}\sum_{i=1}^{G}(r(s, a_i) - \mu)^2} \tag{2.10}$$

$$A^{\pi_{\theta_t}}(s, a_j) = \frac{r(s, a_j) - \mu}{\sigma} \tag{2.11}$$

PPO Objective with Group-Relative Advantages

GRPO then maximizes the standard PPO objective, but using these group-relative advantages instead of value-network-based estimates. The objective becomes:

$$\max_{\theta} \frac{1}{G}\sum_{i=1}^{G} \mathbb{E}_{(s,a_1,\dots,a_G)\sim\pi_{\theta_t}} \left[\begin{cases} \min\left(\frac{\pi_\theta(a_i|s)}{\pi_{\theta_t}(a_i|s)}, 1+\epsilon\right) A^{\pi_{\theta_t}}(s, a_i) & \text{if } A^{\pi_{\theta_t}}(s, a_i) > 0 \\ \max\left(\frac{\pi_\theta(a_i|s)}{\pi_{\theta_t}(a_i|s)}, 1-\epsilon\right) A^{\pi_{\theta_t}}(s, a_i) & \text{if } A^{\pi_{\theta_t}}(s, a_i) < 0 \end{cases} \right] \tag{2.12}$$

This formulation preserves PPO's asymmetric clipping behavior: when advantages are positive (indicating good actions), we clip the importance ratio from above at $(1+\epsilon)$ to prevent over-optimization. When advantages are negative (indicating poor actions), we clip from below at $(1-\epsilon)$ to avoid excessive penalization.

Intuitive Understanding

The intuition behind GRPO is elegantly simple: each policy update makes the model more likely to produce actions that performed relatively better than other actions tried at the same state, and less likely to produce actions that performed relatively worse. This creates a natural competitive dynamic where actions are evaluated against their peers rather than against an absolute baseline.

Consider a concrete example: if for a given coding problem, the model generates five different debugging approaches with rewards $[0.1, 0.8, 0.3, 0.9, 0.2]$, GRPO will:

- Strongly reinforce the action with reward 0.9 (highest z-score)

- Moderately reinforce the action with reward 0.8 (second highest z-score)
- Slightly penalize actions with rewards $0.3, 0.2, 0.1$ (below-average performance)

This relative ranking approach is particularly powerful for code repair where absolute reward values may vary significantly across different types of bugs, but relative solution quality within each problem remains meaningful.

Relationship to PPO

It's crucial to understand that GRPO is not a fundamentally different algorithm from PPO—it is PPO with a specific choice of advantage estimation. The clipping mechanism, importance sampling, and optimization dynamics remain identical. The only change is replacing:

$$\hat{A}_t^{PPO} = R_t - V_\phi(s_t) \tag{2.13}$$

with:

$$\hat{A}_t^{GRPO} = \frac{r_t - \mu_{\text{group}}}{\sigma_{\text{group}}} \tag{2.14}$$

This substitution eliminates the need for:

- Training a separate value network V_ϕ
- Computing value loss $L^{VF}(\phi)$
- Managing value network hyperparameters
- Coordinating policy and value network training schedules

Computational and Practical Advantages

The computational benefits of GRPO are substantial:

Memory Efficiency: Eliminating the value network reduces GPU memory requirements by approximately 50%, enabling larger batch sizes or model sizes within the same hardware constraints.

Training Simplicity: The training loop becomes significantly simpler, reducing implementation complexity and potential sources of bugs. There are no value network updates to coordinate or balance against policy updates.

Hyperparameter Robustness: With fewer moving parts, GRPO exhibits reduced sensitivity to hyperparameter choices, making it more reliable across different tasks and model architectures.

Batch Processing Efficiency: GRPO can naturally handle variable batch sizes and sequence lengths without the complications introduced by value network training, which often requires careful batch construction.

Advantages for Code Repair

GRPO's design makes it particularly well-suited for code repair applications:

Natural Handling of Sparse Rewards: Code repair often produces binary success/failure outcomes or sparse quality metrics. GRPO's relative comparison approach handles this naturally, as the standard score normalization adapts to the reward distribution within each group.

Problem Diversity: Different coding problems require vastly different solution approaches and have different inherent difficulty levels. GRPO's group-relative baseline automatically adjusts to each problem's context, whereas a global value function would struggle to capture this diversity.

Exploration Encouragement: By comparing actions against their immediate peers rather than a global baseline, GRPO encourages exploration of diverse solution strategies, which is crucial for learning robust debugging skills.

Computational Scaling: Code repair training requires processing thousands of agent interactions across diverse repositories and bug types. GRPO's computational efficiency makes this scale of training practically feasible.

The mathematical elegance of GRPO lies in its ability to preserve all of PPO's theoretical guarantees while dramatically simplifying the implementation. For code repair, where relative solution quality matters more than absolute reward prediction, this approach provides an optimal balance of performance, simplicity, and computational efficiency.

TODO: Acknowledge variance collapse as a fundamental challenge in GRPO - the policy gradient term inherently incentivizes reducing output variance, leading to less diverse/exploratory behavior. This is an active area of research. Cite relevant papers on this phenomenon and proposed solutions (entropy regularization, KL penalties, etc.). Discuss implications for code repair where diverse exploration strategies are crucial.

2.3 Agent-Based Programming Environments

The integration of language models with interactive programming environments has opened new possibilities for automated software engineering, enabling models to perform complex, multi-step reasoning tasks.

2.3.1 Coding Agent Frameworks

Modern coding agents combine language models with tool access, allowing them to navigate codebases, execute commands, and iteratively refine solutions.

TODO: Survey existing coding agents: GitHub Copilot Workspace, Cursor, Aider, OpenHands/SWE-Agent. Discuss their architectures, tool sets, and performance. Highlight the distinction between inference-time agents vs training-time integration. Review benchmarks like SWE-Bench where scaffolded agents show 1.8x improvement over prompt-only approaches.

2.3.2 Scaffold Design Philosophy

Agent scaffolds represent the interface between language models and their operating environment, with design choices significantly impacting performance and learning outcomes.

TODO: Define scaffolding in the context of coding agents. Contrast minimalist approaches (basic file I/O, shell commands) vs heavyweight approaches (repository mapping, contextual reasoning, guided workflows). Discuss the ”bitter lesson” perspective: whether minimal assumptions lead to better generalization. Review existing work on tool complexity vs performance trade-offs.

The choice of scaffold complexity represents a fundamental design decision that affects both training dynamics and deployment performance, yet this relationship remains poorly understood.

2.3.3 The Nano-Agent: A Minimalist Approach

The nano-agent represents a deliberate exploration of minimalist scaffold design, embodying the principle that simpler, more general tools may enable better learning outcomes than heavily engineered, task-specific environments.

Design Philosophy

The nano-agent's design is guided by the "bitter lesson" from AI research: methods that leverage computation and learning tend to be more effective in the long run than those that rely on human knowledge and engineering. Rather than providing sophisticated tools and contextual assistance, the nano-agent offers only the most fundamental capabilities needed for code interaction.

This minimalist approach serves several purposes:

- **Transparency for RL training**: Simple tools produce clear, interpretable action sequences that facilitate reward computation and training analysis
- **Generalization potential**: Minimal assumptions about programming languages, frameworks, or development practices should enable broader applicability
- **Computational efficiency**: Lightweight operations reduce overhead in RL training loops where thousands of agent interactions must be processed
- **Research reproducibility**: Simple implementations are easier to replicate and extend by other researchers

Tool Interface

The nano-agent provides exactly two primary tools for interacting with codebases:

Shell Command Execution (`shell(cmd)`): This tool allows the agent to execute terminal commands within a restricted bash environment (rbash). Available commands include standard Unix utilities for navigation and inspection:

- File system navigation: `ls`, `cd`, `pwd`, `find`
- Content inspection: `cat`, `head`, `tail`, `less`

- Text processing: `grep`, `awk`, `sed`, `sort`
- Repository operations: `git log`, `git diff`, `git status`

File Patching (`apply_patch`): This tool enables precise code modifications through a search-and-replace mechanism. The agent specifies:

- Target file path
- Exact text to be replaced (old_content)
- Replacement text (new_content)
- Optional context for disambiguation

Safety and Isolation

Security is paramount when allowing language models to execute arbitrary commands. The nano-agent employs several safety mechanisms:

Restricted Bash (rbash): All shell commands execute within a restricted bash environment that prevents:

- Network access and external communication
- File system access outside the designated workspace
- Process spawning beyond allowed utilities
- Modification of system files or configurations

Sandboxed Execution: Each agent session runs in an isolated container with:

- Limited computational resources (CPU, memory, time)
- No persistent state between sessions
- Comprehensive logging of all actions and outputs

Command Validation: Before execution, all commands undergo validation to ensure they match allowed patterns and don't contain potential exploits.

Comparison to Heavyweight Scaffolds

The nano-agent's minimalist design contrasts sharply with heavyweight coding agents that provide extensive support features:

Repository Understanding: While tools like Aider generate comprehensive repository maps and maintain contextual awareness across files, the nano-agent requires models to build this understanding through direct exploration using basic commands.

Guided Workflows: Heavyweight scaffolds often provide structured interaction patterns, step-by-step guidance, and built-in reasoning frameworks. The nano-agent offers no such assistance, forcing models to develop their own debugging strategies.

Error Handling: Advanced agents may provide sophisticated error recovery, automatic retries, and contextual help. The nano-agent provides only raw command output, requiring models to interpret failures and adapt accordingly.

Code Analysis: While heavyweight scaffolds might include syntax parsing, dependency analysis, and semantic understanding, the nano-agent relies on the model's inherent code comprehension abilities supplemented only by basic text processing tools.

Advantages for RL Training

The nano-agent's simplicity offers several advantages specifically for reinforcement learning applications:

Complete Action Logging: Every agent interaction produces clear, interpretable logs that can be analyzed to understand learning dynamics and failure modes. There are no hidden internal operations or black-box processing steps.

Reward Clarity: With minimal tool complexity, it becomes easier to attribute successes and failures to specific agent decisions, enabling more accurate reward assignment and training signal propagation.

Scalable Batch Processing: Simple operations can be efficiently parallelized across multiple training instances without the overhead of complex state management or resource coordination required by heavyweight scaffolds.

Debugging and Analysis: When training fails or produces unexpected results, the minimal tool set makes it easier to identify root causes and adjust training procedures accordingly.

Expected Learning Dynamics

The hypothesis underlying the nano-agent design is that models trained with minimal scaffolds will develop more robust, generalizable debugging skills. Specifically:

Fundamental Skill Development: Without sophisticated assistance, models must learn essential programming skills like code navigation, pattern recognition, and systematic debugging approaches.

Adaptability: Models that succeed with minimal tools should adapt more readily to new environments, programming languages, and unexpected scenarios.

Self-Reliance: Rather than depending on engineered heuristics and guided workflows, models learn to formulate their own strategies and recover from mistakes independently.

Transfer Learning: Skills learned through fundamental tool use should transfer more effectively to different programming contexts than highly specialized, environment-specific behaviors.

This design philosophy will be empirically tested through direct comparison with heavyweight scaffold approaches, providing insights into the optimal balance between tool sophistication and learning effectiveness in agent-based code repair systems.

2.4 Agent-in-the-Loop Training

The integration of agent frameworks directly into model training represents a paradigm shift from traditional supervised learning approaches, enabling models to learn through active interaction rather than passive observation.

TODO: Define agent-in-the-loop training and contrast with traditional approaches. Explain why this is novel: most prior work trains on static data then deploys as agents. Discuss technical challenges: reward engineering, environment stability, computational scaling. Review limited prior work in this area, highlighting gaps this thesis addresses.

Current research in this area is primarily conducted by industry labs (OpenAI, Anthropic, Cognition Labs) with limited open-source replication, creating a significant knowledge gap in the academic community.

2.4.1 Reward Design for Code Repair

Effective reward design is crucial for agent-in-the-loop training, requiring careful balance between task-specific objectives and general coding principles.

TODO: Discuss reward engineering for code tasks: patch similarity metrics, test execution results, code quality measures. Review challenges: sparse signals, delayed feedback, difficulty capturing semantic correctness. Explain the specific reward formulation used in this work: diff-based similarity to oracle patches, file-level matching, format compliance.

2.5 Evaluation Benchmarks

Rigorous evaluation of automated code repair systems requires diverse, realistic benchmarks that capture the complexity of real-world debugging scenarios.

2.5.1 SWE-Bench Family

The SWE-Bench benchmark series has emerged as the gold standard for evaluating coding agents on realistic software engineering tasks.

TODO: Describe SWE-Bench evolution: original dataset, SWE-Bench-Verified improvements, Multi-SWE-Bench for language diversity. Explain task format: GitHub issues with repository context and ground truth patches. Discuss evaluation methodology and why success rates remain low (20%) even for state-of-the-art systems.

2.5.2 Cross-Language Generalization

Evaluating generalization across programming languages provides insights into whether learned debugging skills transfer beyond training environments.

TODO: Review Java benchmarks like Defects4J v2.0, GitBug-Java. Discuss why cross-language evaluation is important for understanding fundamental reasoning vs language-specific pattern matching. Explain the hypothesis that minimalist scaffolds should generalize better across languages due to fewer language-specific assumptions.

2.6 Related Work

This section synthesizes prior research most directly relevant to our agent-in-the-loop approach, highlighting gaps that this thesis addresses.

TODO: Comprehensive review of: (1) RL for code tasks - limited prior work, mostly focused on code generation rather than repair; (2) Agent-based debugging - mostly inference-time systems, not training integration; (3) Scaffold complexity studies - very limited research on tool complexity vs learning outcomes. Emphasize what's missing: open-source agent-in-loop training, systematic scaffold comparison, monotonic improvement demonstration.

TODO: Conclude by positioning this work: first open-source replication of agent-in-loop RL for code repair, novel comparison of scaffold complexity effects, potential to democratize access to advanced coding agent training techniques previously available only to industry labs.

Chapter 3

Method

This chapter presents our novel approach to training large language models for automated code repair through agent-in-the-loop reinforcement learning. We detail the experimental design, technical implementation, and evaluation methodology used to investigate whether embedding coding agents directly into the RL training loop improves bug-fixing performance.

3.1 Overview of Agent-in-the-Loop Reinforcement Learning

Traditional approaches to training LLMs for code repair rely on supervised fine-tuning with static datasets of code-patch pairs. In contrast, our method pioneers *agent-in-the-loop reinforcement learning*, where coding agents actively interact with real software repositories during training. This paradigm shift transforms models from passive learners observing fixed examples into active agents that learn through environmental interaction and experiential feedback.

The core innovation lies in our integration of existing coding agent frameworks directly into the RL training pipeline. Rather than constraining models to single-pass generation, we enable multi-step interactions where agents can:

- Navigate repository structures using terminal commands
- Examine multiple files to understand code context
- Iteratively refine solutions based on environmental feedback

- Learn from the outcomes of their actions rather than just imitating examples

By implementing an OpenAI-compatible API server with asynchronous token streaming capabilities, we bridge the gap between standard RL training frameworks and agent scaffolding. This enables our custom Nano coding agent to interact naturally with repositories through basic terminal commands while maintaining compatibility with the RL training loop.

TODO: Add a figure illustrating the agent-in-the-loop architecture showing the feedback cycle between the LLM, agent scaffold, code repository, and reward signal

3.2 Experimental Setup

3.2.1 Datasets and Benchmarks

Our experimental design carefully separates training and evaluation data to demonstrate true generalization rather than memorization. We employ the following datasets:

Training Dataset: We use SWE-Gym, a curated collection of approximately 2,400 Python bug-fixing tasks derived from real GitHub repositories. Each task includes:

- A containerized repository snapshot at the time of the bug report
- The original issue description
- The ground-truth patch that resolved the issue
- Isolated execution environments for safe agent interaction

SWE-Gym's containerized design makes it ideal for RL training, as agents can freely explore and modify code without risk while receiving deterministic feedback based on their actions.

Primary Evaluation: SWE-Bench-Verified serves as our main evaluation benchmark, containing approximately 500 carefully validated Python bugs from popular open-source projects. These bugs are notably more challenging than the training set, often requiring multi-file modifications and deep understanding of project structure. This dataset tests whether our agent-trained models can generalize beyond their training distribution.

Generalization Testing: To assess cross-language transfer of learned repair skills, we evaluate on Defects4J v2.0, which contains 835 real Java bugs from mature projects. This tests whether the code repair strategies learned in Python environments transfer to syntactically different languages—directly addressing our research question about the generality of agent-learned skills.

3.2.2 Model Selection

We base our experiments on the Qwen2.5-Coder family of models, specifically the 7B parameter variant. This choice is motivated by:

- Strong baseline performance on code understanding tasks
- Open availability enabling reproducible research
- Sufficient capacity to learn complex agent behaviors while remaining computationally tractable
- Compatibility with our distributed training infrastructure

The model undergoes continued training rather than traditional fine-tuning, preserving its general capabilities while acquiring specialized agent skills through reinforcement learning.

3.3 The Nano Coding Agent Scaffold

Our Nano coding agent represents a minimalist approach to agent scaffolding, designed to provide essential repository interaction capabilities without imposing rigid workflows or excessive complexity. The scaffold enables models to:

3.3.1 Core Capabilities

The Nano agent provides a streamlined set of tools for repository interaction:

- `bash`: Execute shell commands for navigation and file system operations
- `str_replace`: Perform precise string replacements for code modification
- `view_file`: Examine file contents with optional line range specification
- `write_file`: Create new files when necessary

This minimal toolset encourages models to develop their own strategies for code understanding and modification rather than relying on pre-engineered heuristics. The agent must learn to:

- Navigate unfamiliar codebases using standard Unix commands
- Identify relevant files through grep searches and directory exploration
- Understand code context by examining multiple related files
- Apply targeted fixes using precise string replacements

3.3.2 Design Philosophy

The Nano scaffold embodies several key design principles:

Simplicity: By providing only essential tools, we reduce the complexity of the action space and make it easier for models to learn effective strategies through trial and error.

Flexibility: The agent is not constrained to any particular workflow—it can develop its own patterns for approaching different types of bugs based on what proves effective during training.

Transparency: All agent actions map directly to interpretable operations that developers would perform manually, making the learned behaviors more understandable and trustworthy.

Efficiency: The minimal interface reduces computational overhead and allows for faster training iterations compared to more complex scaffolding systems.

3.3.3 Integration with RL Training

The Nano agent integrates seamlessly with our RL training pipeline through a structured action-observation loop:

1. The agent receives an issue description and repository state
2. It generates a sequence of tool calls to explore and understand the codebase
3. Each action produces observations (command outputs, file contents, etc.)
4. The agent iteratively refines its understanding and proposes fixes

5. The final patch is evaluated against the ground truth for reward computation

This cycle allows the model to learn from both successful and unsuccessful repair attempts, gradually improving its ability to navigate codebases and identify correct fixes.

3.4 Reinforcement Learning Training Algorithm

3.4.1 Group Relative Policy Optimization (GRPO)

We employ Group Relative Policy Optimization (GRPO) as our primary RL algorithm. GRPO offers several advantages for our agent-in-the-loop setting:

- **No value model required:** Unlike traditional actor-critic methods, GRPO estimates advantages using relative performance within a batch, eliminating the need for a separate value network
- **Reduced variance:** By normalizing rewards across groups of trajectories, GRPO provides more stable training signals
- **Simplified pipeline:** The absence of value estimation reduces computational overhead and implementation complexity

3.4.2 Training Process

Our training follows an iterative process where the agent attempts to solve bugs from the SWE-Gym dataset:

1. **Trajectory Generation:** For each bug in a training batch, the agent generates a complete trajectory—from initial exploration through final patch submission
2. **Reward Computation:** Each trajectory receives a reward based on the similarity between the generated patch and the ground-truth solution (detailed in Section 3.5)
3. **Advantage Estimation:** GRPO computes advantages by comparing each trajectory's reward to the mean reward within its group
4. **Policy Update:** The model parameters are updated to increase the likelihood of high-advantage actions while maintaining proximity to the reference policy

through KL regularization

3.4.3 Hyperparameters

Key hyperparameters for our GRPO implementation include:

- **Batch size:** 32 trajectories per update
- **Learning rate:** 1×10^{-5} with cosine annealing
- **KL coefficient:** $\beta = 0.1$ to balance exploration and stability
- **Maximum trajectory length:** 8192 tokens to accommodate multi-step agent interactions
- **Training iterations:** 5,000 episodes covering the SWE-Gym dataset

These values were selected through preliminary experiments to ensure stable convergence while allowing sufficient exploration of the action space.

3.4.4 Distributed Training Infrastructure

To handle the computational demands of agent-in-the-loop RL, we implement a distributed training architecture:

- **DeepSpeed ZeRO-3:** Model parameters, gradients, and optimizer states are sharded across GPUs to enable training of 7B parameter models
- **Asynchronous trajectory collection:** Multiple agent instances run in parallel, maximizing GPU utilization
- **Dynamic batching:** Trajectories of varying lengths are efficiently packed to minimize padding overhead

This infrastructure enables us to train on realistic, multi-step agent interactions while maintaining reasonable training times.

3.5 Reward Design

3.5.1 Outcome-Based Patch Similarity

Our reward function focuses on the final outcome—the generated patch—rather than intermediate steps. This design choice offers several benefits:

- **Simplicity:** Evaluating final patches is straightforward and deterministic
- **Flexibility:** Agents can discover diverse problem-solving strategies without being constrained by process-specific rewards
- **Alignment:** The reward directly measures what we care about—correct bug fixes

3.5.2 Reward Computation

For each generated patch, we compute rewards based on similarity to the ground-truth solution:

$$R(p_{\text{gen}}, p_{\text{true}}) = \begin{cases} 1.0 & \text{if } p_{\text{gen}} = p_{\text{true}} \\ \text{sim}(p_{\text{gen}}, p_{\text{true}}) & \text{if partial match} \\ 0.0 & \text{otherwise} \end{cases} \tag{3.1}$$

Where sim measures line-level similarity between patches, awarding partial credit for fixes that modify the correct locations but with imperfect changes.

3.5.3 Addressing Sparse Rewards

The sparse nature of exact patch matching presents challenges for RL training. We address this through:

- **Curriculum learning:** Starting with simpler bugs where rewards are more attainable
- **Partial credit:** Rewarding patches that target correct files and functions even if the exact fix differs

- **Large batch sizes:** Ensuring sufficient positive examples within each training batch

3.5.4 Future Extensions

While our current approach uses patch similarity for computational efficiency, the framework naturally extends to test-based evaluation. Future work could incorporate:

- Execution of project test suites to verify functional correctness
- Multi-objective rewards balancing correctness, code quality, and efficiency
- Human preference learning for subjective aspects of code style

These extensions would require significant infrastructure investment but could lead to more robust and generalizable repair capabilities.

3.6 Training Infrastructure: vLLM Integration

3.6.1 OpenAI-Compatible API Server

A key technical innovation in our approach is the integration of vLLM's asynchronous serving capabilities with the RL training loop. This enables:

- **Seamless agent integration:** Existing coding agents designed for OpenAI's API can be used without modification
- **Asynchronous token streaming:** Real-time generation allows for natural multi-turn interactions
- **Parallel trajectory collection:** Multiple agent instances can explore different solutions simultaneously

3.6.2 Architecture Overview

Our training infrastructure consists of several interconnected components:

1. **vLLM Inference Server:** Hosts the model and provides OpenAI-compatible endpoints

2. **Agent Workers:** Multiple Nano agent instances that interact with code repositories
3. **Trajectory Collectors:** Gather complete agent episodes for batch processing
4. **GRPO Trainer:** Computes rewards and updates model parameters
5. **Weight Synchronization:** Ensures all components use the latest model weights

3.6.3 Real-Time Weight Updates

Unlike traditional RL setups where inference and training are separate phases, our system enables continuous learning:

- Model weights are updated after each batch of trajectories
- Updates are immediately synchronized to all inference processes
- Agents benefit from improvements within the same training session

This tight integration between serving and training represents a significant departure from conventional approaches, enabling more efficient exploration and faster convergence.

3.6.4 Implementation Details

Key implementation choices include:

- **Collective RPC:** For efficient weight sharing across distributed processes
- **LoRA adaptation:** Optional use of low-rank adapters to reduce communication overhead
- **Containerized environments:** Each agent runs in an isolated Docker container for safety
- **Request batching:** Multiple agent requests are processed concurrently for efficiency

TODO: Add a diagram showing the data flow between vLLM server, agent workers, and GRPO trainer

3.7 Evaluation Methodology

3.7.1 Evaluation Metrics

We assess model performance using several complementary metrics:

- **Success Rate:** Percentage of bugs for which the agent generates an exactly correct patch
- **Partial Success Rate:** Including fixes that target the correct location with minor differences
- **Time to Solution:** Average wall-clock time required to generate a fix
- **Exploration Efficiency:** Number of files examined and commands executed per bug

3.7.2 Baseline Comparisons

To contextualize our results, we compare against several baselines:

1. **Base Model:** The pretrained Qwen2.5-Coder without any bug-fixing training
2. **Supervised Fine-Tuning:** The same model fine-tuned on bug-fix pairs using standard supervised learning
3. **Direct Generation:** Models prompted to generate fixes without agent scaffolding
4. **State-of-the-art Systems:** Published results from other automated repair approaches

3.7.3 Generalization Testing

Beyond in-domain performance, we evaluate generalization through:

- **Cross-dataset evaluation:** Testing on SWE-Bench-Verified after training on SWE-Gym
- **Cross-language evaluation:** Applying Python-trained models to Java bugs
- **Temporal generalization:** Testing on bugs from time periods not covered in training

3.7.4 Ablation Studies

To understand the contribution of different components, we conduct ablations:

- **Without RL:** Using only supervised fine-tuning on the same data
- **Without agent scaffold:** Direct patch generation without repository interaction
- **Varying trajectory lengths:** Impact of allowing more or fewer exploration steps
- **Different reward functions:** Comparing patch similarity vs. binary success rewards

3.7.5 Statistical Significance

All reported improvements are tested for statistical significance using:

- Bootstrap confidence intervals for success rate differences
- McNemar's test for paired comparisons on the same test set
- Effect size measurements (Cohen's d) to quantify practical significance

This comprehensive evaluation framework ensures that our conclusions about agent-in-the-loop RL are well-supported and reproducible.

Chapter 4

<The work>

Describe the degree project. What did you actually do? This is the practical description of how the method was applied.

Chapter 5

<Result>

Describe the results of the degree project.

Chapter 6

<Conclusions>

Describe the conclusions (reflect on the whole introduction given in Chapter 1).

Discuss the positive effects and the drawbacks.

Describe the evaluation of the results of the degree project.

Describe valid future work.

The sections below are optional but could be added here.

6.1 Discussion

6.1.1 Future Work

6.1.2 Final Words

Appendix - Contents

Appendix A

First Appendix

This is only slightly related to the rest of the report

Appendix B

Second Appendix

this is the information