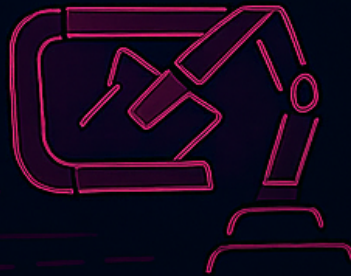
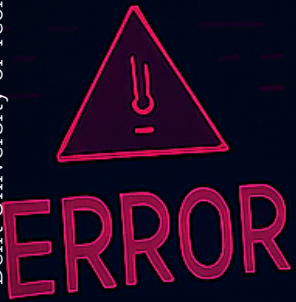


Fixing on the Fly

Adapting Large Language Models to
Domain-Specific Automatic Program Repair

IN5000: Final Project

Ana Terna



OK ✓

Fixing on the Fly

Adapting Large Language Models to Domain-Specific Automatic Program Repair

by

Ana Terna

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended on 27th of January, 2026

Student Number: 6091253
Project Duration: July, 2025 - January, 2026
Thesis committee: Prof. dr. Arie van Deursen, TU Delft
Daily Supervisor: Assistant Prof. dr. Maliheh Izadi, TU Delft
Core Member: Assistant Professor. dr. Jie Yang. TU Delft
Company mentors: Timur Galimzyanov, JetBrains, AI researcher, Code Modelling team
Sergey Titov, JetBrains, Team Lead, Code Modelling team

Faculty: Faculty of Electrical Engineering, Mathematics and Computer Science, Delft

Cover: Sora generated cover
Style: TU Delft Report Style, with modifications by Daan Zwaneveld

Preface

I chose to pursue this Master's degree at TU Delft because it had been part of my plan since I left Moldova to study in the Netherlands, at a time when I had a limited understanding of computer science as a discipline. Even then, I believed that learning to code would allow me to build concrete solutions to real problems. However, after completing my Bachelor's and starting the Master's program, I started to question whether advancing further in theory alone would truly help me achieve this goal, beyond applying the practical skills I had already acquired.

Looking back, I now recognize that many lectures by exceptional teachers shaped my way of thinking more deeply than I had expected. At the same time, I joined Booking.com as a full-time software engineer, where I worked on cloud computing and payment systems. This professional experience shaped my academic studies and played a central role in improving both my technical skills and my mindset as an engineer.

A decisive turning point came with the course *Machine Learning for Software Engineering*, taught by Maliheh Izadi. In this course, I was introduced to the foundations of large language models, including how they are designed, trained, validated, and applied. This subject immediately captured my interest and revealed a new path for me, one that I had not anticipated but quickly embraced. From that moment, I knew I would be happy to pursue my thesis under her supervision, and when the opportunity arose to collaborate with JetBrains, my decision was clear.

I am deeply grateful to Maliheh Izadi for her continuous guidance and support throughout this journey and for indirectly inspiring me to explore this new domain. I also thank Prof. Arie van Deursen for serving as my thesis advisor and for his valuable feedback, and Jie Yang for participating in the evaluation committee.

At JetBrains, I conducted this research under the supervision of Sergey Titov, team lead of the Code Modeling team, whom I thank for offering me this opportunity and for giving me the freedom to shape this work. My main mentor was Timur Galimzyanov, with whom I had the pleasure of closely collaborating on this work. I am sincerely grateful to Timur for his patience, his extensive knowledge, and his constant support. Beyond his technical guidance, I especially appreciated the many conversations in which he patiently helped me practise Russian with him and the team! I am also thankful for his willingness to answer my never-ending questions about artificial intelligence.

None of this would have been possible without my parents. I thank my mother for always believing in me and encouraging me to take on greater challenges, and my father for teaching me to remain true to myself and to trust the process.

Finally, I would like to thank Alex, my partner and constant source of support, with whom I share both the difficulties and the joy of growing in the same field together.

*Ana Terna
Delft, January 2026*

Summary

Automated program repair (APR) is increasingly critical in modern software development, yet language models (LMs) often struggle to capture repository-specific conventions and constraints. Small language models (SLMs) offer a cost-effective and deployable alternative, but their performance depends heavily on high-quality domain-specific supervision. In this work, we introduce a multi-teacher distillation pipeline that generates multi-turn repair trajectories, including both successful fixes and intermediate failures, to construct rich training datasets for method-level APR. We systematically analyze the impact of dataset size, repair diversity, fine-tuning strategies, hyperparameters, and reasoning supervision, aiming to identify efficient and reliable approaches for adapting SLMs to repository-specific repair tasks.

Our experiments demonstrate that parameter-efficient fine-tuning, particularly LoRA with carefully selected adapter ranks, achieves strong performance across reasoning and non-reasoning regimes while maintaining low computational cost. Explicit reasoning supervision is not required for high repair accuracy, but it significantly reduces reasoning trace lengths and inference costs. Dataset diversity and multi-turn trajectories are key to improving generalization and bridging the gap between reasoning and non-reasoning inference. Finally, this study seeks to provide empirical insights into the practical adaptation of SLMs for repository-specific APR, evaluating how strategic choices in dataset design, lightweight fine-tuning approaches, and reasoning supervision influence performance in real-world contexts.

Contents

| | |
|---|-----------|
| Preface | i |
| Summary | ii |
| Nomenclature | v |
| 1 Introduction | 1 |
| 1.1 Research Questions | 2 |
| 1.2 Thesis Outline | 3 |
| 2 Background | 4 |
| 2.1 Large Language Models | 4 |
| 2.2 Rise of LLMs for Code Tasks | 5 |
| 2.2.1 Capabilities | 5 |
| 2.2.2 Benchmarks | 5 |
| 2.2.3 Trends in LLM-Driven Development | 6 |
| 2.2.4 Domain Adaptation and Specialization | 6 |
| 2.3 Knowledge Transfer in LLMs | 6 |
| 2.3.1 Prompting and Retrieval-Augmented Generation | 6 |
| 2.3.2 Fine-tuning | 7 |
| 2.3.3 Reinforcement Learning | 7 |
| 2.3.4 Dataset Construction | 7 |
| 2.3.5 Knowledge Distillation | 8 |
| 2.3.6 Reasoning | 8 |
| 2.4 Related Work | 9 |
| 2.4.1 Fine-Tuning Strategies for APR | 9 |
| 2.4.2 Multi-Turn and Agentic Repair with Execution Feedback | 9 |
| 2.4.3 Limitations of Current Approaches | 10 |
| 3 Methodology | 11 |
| 3.1 Repository Selection and Data Extraction | 12 |
| 3.2 Multi-Teacher Knowledge Distillation | 13 |
| 3.2.1 Teacher Model Selection | 13 |
| 3.2.2 Trajectory Generation | 13 |
| 3.3 Reasoning Distillation | 14 |
| 3.3.1 Eliciting Reasoning from Teacher Models | 14 |
| 3.3.2 Addressing Reasoning Length and Efficiency | 15 |
| 3.3.3 Single-Turn Reasoning Distillation | 16 |
| 3.3.4 Hybrid Training with Reasoning and Direct Repair | 16 |
| 4 Experimental Setup | 17 |
| 4.0.1 Benchmark | 17 |
| 4.0.2 Computational Infrastructure | 17 |
| 4.1 Supervised Fine-Tuning Strategies | 17 |
| 4.1.1 Functional Correctness Metrics | 19 |
| 4.1.2 Reasoning Quality Assessment | 19 |
| 4.1.3 Baseline and Generalization Assessment | 19 |
| 5 Results | 21 |
| 5.1 Baseline Performance | 21 |
| 5.2 Supervised Fine-Tuning Without Reasoning | 22 |
| 5.2.1 Dataset Evolution and Multi-Teacher Distillation | 22 |

| | | |
|----------|--|-----------|
| 5.2.2 | LoRA versus Full Fine-Tuning | 22 |
| 5.2.3 | Hyperparameter Analysis: LoRA Rank Selection | 25 |
| 5.3 | Reasoning-Augmented Fine-Tuning | 26 |
| 5.3.1 | Reasoning Quality and Teacher Alignment | 29 |
| 5.3.2 | Token Efficiency Gains | 30 |
| 5.4 | Error Analysis | 31 |
| 5.5 | Generalization Across Model Families | 31 |
| 5.5.1 | Out-of-Domain Performance: SWE-bench | 32 |
| 6 | Discussion | 33 |
| 6.1 | Answer to research questions | 33 |
| 6.1.1 | Factors for Efficient and Cost-Effective In-Domain APR Fine-Tuning | 33 |
| 6.1.2 | Effects of Reasoning Supervision on Repair Performance | 34 |
| 6.2 | Limitations and Threats to Validity | 34 |
| 6.3 | Future Work: Context-Aware Training and Complementary Tasks | 35 |
| 7 | Conclusion | 36 |
| | References | 37 |
| A | Prompting Setup | 41 |
| A.1 | Prompt Templates Used in the Django APR Benchmark | 41 |
| A.2 | Reasoning Prompts Considered | 41 |
| B | Additional Results | 43 |
| B.1 | Fine-tuning results of additional experiments | 43 |
| B.2 | The frequency of reasoning tokens across models. | 46 |
| C | Output Model examples | 49 |

Nomenclature

Abbreviations

| Abbreviation | Definition |
|-----------------|---|
| APR | Automated Program Repair |
| CoT | Chain of Thought |
| DA | Domain Adaptation |
| FFT | Full Parameter Fine-Tuning |
| GPT | Generative Pre-trained Transformer |
| IDE | Integrated Development Environment |
| KD | Knowledge Distillation |
| LM | Language Model |
| LLM | Large Language Model |
| LoRA | Low-Rank Adaptation |
| MBPP | Mostly Basic Python Problems (benchmark) |
| PEFT | Parameter-Efficient Fine-Tuning |
| QLoRA | Quantized Low-Rank Adaptation |
| RAG | Retrieval-Augmented Generation |
| RL | Reinforcement Learning |
| SE | Software Engineering |
| SFT | Supervised Fine-Tuning |
| SLM | Small Language Model |
| SOTA | State of the Art |
| SWE | Software Engineering Evaluation (used in SWE-bench and SWE-agent) |
| <i>think</i> | Training regime with reasoning-augmented repair traces |
| <i>no-think</i> | Training regime using direct code fixes only, without reasoning |

List of Tables

| | | |
|-----|--|----|
| 3.1 | Evolution of training datasets through incremental addition of teacher trajectories and synthesis strategies. Each version builds upon previous versions, incorporating new teacher models, ground truth augmentation, and task diversity. | 14 |
| 4.1 | Hyperparameter configurations for LoRA and full fine-tuning. Shared parameters apply to both methods. | 18 |
| 5.1 | Baseline results on the Django benchmark in <i>think</i> and <i>no-think</i> modes. The table shows that model performance scales with size and that reasoning consistently improves accuracy, showing the performance gap that our fine-tuning methods aim to reduce. | 21 |
| 5.2 | Performance evolution across dataset versions for <i>Qwen3-8B</i> . Dataset size increases from 1052 (<i>v1</i>) to 14051 (<i>v5</i>) examples. LoRA achieves peak <i>think</i> mode performance with <i>v4</i> (82.4%, +5.9 points), while FFT peaks in <i>no-think</i> mode with <i>v5</i> (79.6%, +9.7 points), nearly closing the <i>think-no-think</i> gap. | 22 |
| 5.3 | Performance of reasoning-augmented models compared to the best non-reasoning baseline (<i>v4</i> , LoRA $r = 32$). Hybrid models combine reasoning and nothink examples. The experiment name is composed of <base model>-reasoning-<teacher size>-teacher-<dataset>-<specification>. | 26 |
| 5.4 | Alignment of student models with teacher reasoning, measured using <i>ROUGE - L</i> , <i>Jaccard</i> , and <i>CodeBLEU</i> . Scores are reported relative to the base model (<i>Qwen3-8B</i>) and teacher model (<i>Qwen3-32B</i>). Higher values indicate greater similarity to the teacher. The <i>Qwen3-8B</i> base model serves as the foundation for all fine-tuned variants. | 29 |
| 5.5 | <i>CodeBLEU</i> and <i>ExactMatch</i> of model-generated methods compared to ground-truth Django implementations. Model names follow the format <base model>-<dataset>-<reasoning approach> (fine-tuning method). Higher values indicate better alignment with the reference implementations. | 30 |
| 5.6 | Reasoning token statistics across training configurations (<i>think</i> -mode inference on the benchmark). Percentage reductions relative to the base model are shown in parentheses. | 31 |
| 5.7 | APR distillation generalizes across model families with gains inversely proportional to model size (smaller models benefit most). All models are trained with LoRA ($r = 32$) on <i>v4</i> dataset. | 32 |
| 5.8 | Out-of-domain repair on 220 Django issues from SWE-bench. LoRA preserves most of the base model’s performance, while FFT induces substantial catastrophic forgetting in the agentic setting. | 32 |

List of Figures

| | | |
|-----|--|----|
| 3.1 | Overview of the knowledge distillation pipeline for domain-specific program repair. Teacher models generate multi-turn trajectories on repository methods, which are distilled into training datasets combining repair examples, reasoning traces, and single-turn generation tasks. The student model is then fine-tuned using either LoRA or full fine-tuning under <i>think</i> and <i>no-think</i> regimes. | 11 |
| 3.2 | The distribution of characters in the reasoning blocks distilled from <i>Qwen3-8B</i> for repair trajectories (<i>v4</i> dataset). | 15 |
| 3.3 | The distribution of characters in the reasoning blocks distilled from <i>Qwen3-32B</i> for repair trajectories (<i>v4</i> dataset). | 15 |
| 3.4 | The distribution of characters in the reasoning blocks distilled from <i>Qwen3-235B-A22B-Thinking-2507</i> for repair trajectories (<i>v4</i> dataset). | 15 |
| 3.5 | The distribution of characters in the reasoning blocks distilled from <i>Qwen3-32B</i> and shortened with <i>GPT-5-mini</i> for repair trajectories (<i>v4</i> dataset). | 15 |
| 4.1 | Pass@k performance of <i>Qwen3-8B</i> on Django across five evaluation runs in <i>thinking</i> mode. | 20 |
| 4.2 | Pass@k performance of <i>Qwen3-8B</i> on Django across five evaluation runs in <i>no-think</i> mode. | 20 |
| 5.1 | Training dynamics of LoRA and FFT on the <i>v4</i> dataset. Bars show single-turn accuracy at each checkpoint, while lines show repair performance across turns on the Django benchmark. Both approaches improve overall accuracy, with different performance and repair patterns across inference regimes. | 23 |
| 5.2 | Training dynamics of LoRA and FFT on the <i>v5</i> dataset, which includes both method generation and repair trajectories. Bars show single-turn accuracy at each checkpoint, and lines show cumulative repair performance across turns on the Django benchmark. | 24 |
| 5.3 | Impact of LoRA rank on training stability and final performance across <i>v4</i> (repair-only, top row) and <i>v5</i> (repair with method generation, bottom row) datasets. Lower ranks (16–32) converge reliably within 10 epochs. Mid-range rank (64) provides stable nothink performance. Higher ranks (128–256) consistently underperform due to overfitting. | 25 |
| 5.4 | LoRA fine-tuning with repair items (<i>v4</i>) and with added distilled reasoning from <i>Qwen3-32B</i> in <i>thinking</i> (left) vs <i>no-thinking</i> (right) mode. In <i>thinking</i> mode, we notice unstable performance in method generation, while the <i>no-think</i> regime was not affected. | 27 |
| 5.5 | Right: LoRA SFT with repair items (<i>v4</i> dataset) and with added shortened distilled reasoning from <i>Qwen3-32B</i> . Left: LoRA SFT with repair items and method generation (<i>v5</i> dataset) with added distilled reasoning from <i>Qwen3-32B</i> | 27 |
| 5.6 | Benchmark performance across checkpoints after LoRA SFT with distilled reasoning from <i>Qwen3-235B-A22B-Thinking-2507</i> and trained on <i>Qwen3-8B</i> base model. The bar plots represent the score of the model at the respective training epoch, while the line plots represent the score obtained in subsequent turns where the model had to repair. Training with reasoning distilled from a significantly larger teacher model resulted in instability of the model during method generation. Adding slightly diverse reasoning: method generation and repair (right figure), allowed the model to improve slightly against the baseline. | 28 |
| 5.7 | Comparison of LoRA SFT with distilled reasoning. Left: <i>v1</i> repair dataset using primed reasoning from <i>Qwen3-8B</i> . Right: <i>v4</i> repair dataset with distilled reasoning from <i>GPT-5-mini</i> evaluated in <i>thinking</i> mode. | 29 |

| | | |
|-----|--|----|
| 5.8 | Breakdown of unresolved errors (after turn 5) by exception type: base student <i>Qwen3-8B</i> , base teacher <i>Qwen3-32B</i> , <i>Qwen3-8B</i> trained on <i>v4</i> non-reasoning dataset, and <i>Qwen3-8B</i> SFTed on <i>v4</i> reasoning dataset. | 31 |
| B.1 | Checkpoint scores and performance across turns of the <i>Qwen3-8B</i> model SFTed with LoRA on <i>v1</i> dataset across 10 epochs in two regimes: thinking (left) and no-think (right). | 43 |
| B.2 | Comparison of LoRA (<i>v3</i> dataset) and full fine-tuning (<i>v1</i> dataset) across <i>thinking</i> and <i>no-thinking</i> regimes. The bar plots represent the scores of each individual checkpoint during training while the line plots represent the performance in subsequent turns. | 44 |
| B.3 | Comparison of full fine-tuning (<i>v3</i> dataset) across <i>thinking</i> and <i>no-thinking</i> regimes for <i>Qwen3-8B</i> and the performance of <i>Mistral-8B-Instruct-2410</i> trained with LoRA and FFT. The bar plots represent the scores of each individual checkpoint during training while the line plots represent the performance in subsequent turns. | 45 |
| B.4 | Checkpoint scores and performance across turns of the <i>Llama-3.1-8B-Instruct</i> model SFTed with LoRA (left) and <i>Qwen3-14B</i> (right) on <i>v5</i> dataset across 10 epochs. | 46 |
| B.5 | The distribution of reasoning characters during evaluation on the Django benchmark of <i>Qwen3-8B</i> trained with reasoning distilled from <i>Qwen3-32B</i> | 46 |
| B.6 | The distribution of reasoning characters during evaluation on the Django benchmark of <i>Qwen3-8B</i> trained with shorter reasoning distilled from <i>Qwen3-32B</i> | 47 |
| B.7 | The distribution of reasoning characters during evaluation on the Django benchmark of <i>Qwen3-8B</i> trained with reasoning distilled from <i>Qwen3-32B</i> and blended without the reasoning trajectories. | 47 |
| B.8 | The distribution of reasoning characters during evaluation on the Django benchmark of <i>Qwen3-8B</i> trained with reasoning distilled from <i>Qwen3-235B-A22B-Thinking-2507</i> | 48 |

1

Introduction

Large language models (LLMs) are increasingly integrated into software development workflows as code assistants, supporting tasks such as code generation, refactoring, and documentation. Despite their effectiveness, a persistent limitation remains: a non-trivial portion of generated code exhibits semantic errors or fails to satisfy project-specific requirements, requiring substantial manual correction [35, 63]. As a result, developer effort often shifts from writing code to validating, debugging, and repairing model outputs. Automatic Program Repair (APR) directly targets this problem by automating parts of the debugging and patch-generation process, with the goal of reducing post-editing cost and improving development efficiency.

Recent progress in LLM-based APR shows that models pretrained on large code corpora can propose plausible fixes. However, such models are typically optimized for general-purpose use and do not capture repository-specific constraints, including coding conventions, implicit design assumptions, or test-driven behavior. Consequently, generated patches may appear correct in isolation but fail when evaluated against project tests or integrated into the broader codebase. This limitation is particularly relevant in industrial settings, where correctness and maintainability are essential and domain-specific data is often scarce.

At the same time, there is growing interest in small language models (SLMs) for software engineering tasks. Compared to large foundation models, SLMs are cheaper to deploy, easier to fine-tune, and better suited for resource-constrained or on-premise environments common in industry [6]. Throughout this work, we use the term *LLMs* to refer to large models with hundreds of billions to trillions of parameters, and *SLMs* to refer to smaller models with only a few billion parameters. While SLMs offer practical advantages, their performance depends heavily on access to high-quality, domain-specific supervision, which is rarely available in sufficient quantity.

Although reinforcement learning (RL) has been explored as a way to improve code generation and repair [16, 34, 70], setting up RL pipelines in practice is challenging. It requires carefully designed reward functions, stable environments, and extensive compute, which limits its adoption in industrial contexts. As a result, supervised fine-tuning (SFT) remains a practical and widely used approach for adapting models to specific domains. However, there is limited work that systematically studies APR through SFT by analyzing the full fine-tuning pipeline [22], including dataset construction, dataset size, fine-tuning method, hyperparameters, and the role of reasoning supervision. This lack of analysis makes it difficult to understand which design choices are most impactful in practice.

In this work, we address these gaps by proposing a data synthesis and fine-tuning pipeline tailored to repository-constrained settings. Our approach leverages stronger teacher models to generate multi-turn repair trajectories for Python methods. A repair trajectory is defined as a structured sequence consisting of: (i) method-generation context, including class-level information, method signature, and natural-language description; (ii) buggy method implementation; (iii) failed test feedback; and (iv) the corrected solution that satisfies the tests.

We distill such trajectories into SLMs to improve their ability to perform context-aware, test-driven repair. In addition, we study the role of explicit reasoning by comparing two training regimes: a *thinking* regime, where models are trained on reasoning-augmented repair traces, and a *no-thinking* regime, where supervision consists only of direct code fixes. This comparison allows us to quantify the trade-offs between reasoning supervision, performance, and cost, and to evaluate whether reasoning-aware fine-tuning provides benefits beyond strong non-reasoning SFT.

To that end, this paper makes the following contributions:

- We propose a dataset construction pipeline for repository-specific APR, combining unit test outcomes with synthetic multi-turn repair trajectories to enhance SLM performance.
- We study knowledge distillation from larger models, transferring both final fixes and intermediate reasoning signals.
- We analyze trade-offs in fine-tuning SLMs, providing practical guidance to balance cost and performance in research and industry.

1.1. Research Questions

The goal of this work is to investigate how SLMs can be effectively fine-tuned for domain-specific program repair. To this end, we define the following research questions to guide our study:

- **RQ1: What factors are critical for building an efficient and cost-effective fine-tuning pipeline for in-domain APR?**

This question focuses on the core components required for effective in-domain APR training and assesses the effectiveness of our proposed approach. We focus on two central aspects of the pipeline: the construction of high-quality training data and the choice of fine-tuning strategy.

1. **RQ1.1: How do dataset size and multi-teacher distillation during fine-tuning affect the repair performance of SLMs?**

To answer this question, we introduce a dataset construction pipeline based on multi-teacher distillation using repository-internal methods and associated test cases, and evaluate how increasing dataset scale and diversity influence post-SFT repair accuracy and stability among various SLM families.

2. **RQ1.2: How do LoRA and full fine-tuning compare in terms of performance and training cost?**

To address this question, we compare LoRA [21] and full fine-tuning across increasing dataset sizes, analysing their repair performance, convergence behaviour, and computational cost. We further examine the effect of the LoRA rank to characterize the trade-off between model capacity and training efficiency, and analyse multi-turn repair performance to assess robustness across inference turns.

- **RQ2: What is the effect of explicit reasoning supervision on repair performance, training stability, and inference efficiency for in-domain APR fine-tuning?**

Reasoning plays an important role in program repair by enabling backtracking, reflection, and self-correction, which motivates our study of reasoning-aware fine-tuning using a distillation-based setup. This question examines the role of reasoning supervision in APR. We analyze whether explicit reasoning improves repair performance and how it affects training and inference cost. In particular, we study how reasoning distilled from teacher models of different families and sizes influences the stability of the base model. Motivated by the overthinking behavior observed in SLMs, we experiment with shorter reasoning traces and with mixtures of reasoning and non-reasoning training data. Our goal is to reduce reasoning length without degrading performance, and to assess whether fine-tuning can narrow the gap between reasoning and non-reasoning inference regimes.

1.2. Thesis Outline

This thesis is organized as follows. Chapter 2 provides the necessary background on APR and language models, and reviews related work to situate this research within the existing literature. Chapter 3 presents the proposed methodology, including the multi-teacher data synthesis pipeline and the SFT strategies used to adapt SLMs to repository-specific repair tasks. Chapter 4 describes the experimental setup, including the datasets, model configurations, and evaluation metrics. Chapter 5 reports the empirical results, analyzing the effects of dataset composition, fine-tuning strategy, hyperparameter choices, reasoning supervision, and generalization across models and tasks. Chapter 6 discusses these findings, the limitations of the study, and directions for future research. Finally, Chapter 7 summarizes the main contributions of this thesis.

2

Background

This section reviews prior research that frames the context of our work, with a focus on developments in LLMs for coding-related tasks. We highlight current trends in the field and discuss how these approaches align with the broader objective of improving the reliability and effectiveness of automated program repair.

2.1. Large Language Models

LLMs have fundamentally reshaped expectations about the future of work and, more broadly, the role of AI in human society. Their widespread impact raises the question of what makes these models so distinctive compared to earlier approaches. In essence, LLMs are the product of decades of progress in machine learning, culminating in the deep learning breakthrough that demonstrated how large neural architectures trained on massive datasets could outperform traditional methods. Geoffrey Hinton, a pioneer of this revolution, is often referred to as the “Godfather of AI” for his foundational contributions to neural network research¹ and he has been an active public voice on both its potential and implications. LLMs are neural networks trained on large text corpora using self-supervised learning. The training approach consists of autoregressively learning to predict the next token in a sequence based solely on the preceding tokens. This left-to-right prediction process supports sequential text generation, allowing the model to produce coherent output one token at a time.

While different architectures can support these models, this work focuses specifically on causal decoder-only models. These models use masked self-attention to ensure that each token can only attend to previous ones, aligning directly with the autoregressive objective. Unlike encoder-only models, which use bidirectional context and are not suitable for generation, or encoder-decoder models, which use a separate decoder for generation, causal decoders unify training and generation under the same mechanism, thereby achieving benefits such as reduced costs. This architecture is currently employed in many publicly accessible models used in real-world applications and serves as the basis for all models studied in this work.

Beyond natural language understanding tasks, the same autoregressive modeling approach applies naturally to software engineering (SE), where code can be treated as a structured form of text. Causal decoder-only models, when pretrained on large corpora of source code, have shown strong performance in code-related tasks. This includes code synthesis, repair, and completion, where the sequential nature of code tokens aligns well with left-to-right generation [20]. A notable example is Codex [10], a model fine-tuned specifically for programming tasks, which has influenced modern software development by assisting with debugging, refactoring, and routine coding.

¹<https://discover.research.utoronto.ca/26059-geoffrey-e-hinton>

2.2. Rise of LLMs for Code Tasks

A defining characteristic of the current LLM ecosystem is the divide between openly released models and those kept proprietary. On the open side, providers such as Meta with the LLaMA family [17], Mistral AI with the Mistral series [26], and Alibaba Cloud with the Qwen squad [57] have adopted licensing strategies that allow model weights to be downloaded and adapted for downstream use. By releasing model weights openly, providers have enabled rapid community-driven progress, facilitating academic research, reproducibility, and domain-specific fine-tuning [38]. DeepSeek AI has further advanced this trend by releasing models that experiment with reward modeling-based fine-tuning to reduce training costs and improve its performance [12]. In contrast, models such as OpenAI’s GPT series [42], Anthropic’s Claude family [4], and Google DeepMind’s Gemini [51] are available solely via proprietary APIs (i.e., closed source), yet they reliably deliver state-of-the-art (SOTA) results across a broad spectrum of tasks. Concurrently, active research efforts aim to reach comparable levels of performance, with steady advances that progressively close the gap to these closed-source systems [12].

2.2.1. Capabilities

When applied to programming, LLMs demonstrate a wide range of capabilities. At their core, these models can perform **code synthesis**, producing syntactically correct and semantically meaningful implementations given a natural language prompt [10, 14]. LLMs are also effective in **code summarization** and **explanation**, where they generate human-readable documentation or translate source code into natural language [49]. Another emerging capability is **code refactoring**, where models restructure code to improve readability, maintainability, or style while preserving functionality [11]. In the context of testing, LLMs have been used for **test case generation** and even for **bug reproduction** based on natural language bug reports [30]. In addition, LLMs have been integrated with static analysis or runtime feedback to facilitate **bug detection** [39]. LLMs are also increasingly utilized for **interactive programming assistance**, enabling iterative, multi-turn interactions with developers to refine code, offer feedback, and adjust solutions [28]. This, in turn, naturally connects to **Automated Program Repair** [22], where LLMs seek to produce concrete patches that can be validated or further refined before being merged into the system.

2.2.2. Benchmarks

Systematic evaluation is crucial for identifying which models can produce code that is both syntactically valid and functionally correct, and a wide range of benchmarks has been developed to measure diverse SE capabilities as well as more specialized applications [31]. Several of these benchmarks focus on code generation using unit test-based assessment. For Python in particular, HumanEval [10] has emerged as the most commonly used benchmark, with adaptations now available for other languages such as Java, JavaScript, and C++ [68, 8]. Complementing HumanEval, EvalPlus [35] enhances the evaluation protocol by providing more exhaustive validation of generated programs, while the MBPP benchmark [5] extends coverage with a larger and more varied set of synthesis tasks.

However, pure code generation from natural language instructions fails to reflect the full range of real developer workflows. To better approximate these interactions, LiveCodeBench [24] evaluates a broader set of programming-related skills, including self-repair and test-output prediction in competitive programming settings. Still, these tasks are largely self-contained. To more closely mirror real-world development, SWE-bench [28] evaluates LLMs on resolving GitHub issues, requiring changes across multiple files and tackling more complex bugs, thus more accurately representing practical software maintenance challenges. In the Java ecosystem, Defects4J [29] was introduced to support similar assessments of automated repair capabilities. The reasoning abilities of LLMs have also been examined. For instance, CruxEval evaluates whether models can reason about the execution behavior of simple Python programs, showing that although chain of thought (CoT) prompting and fine-tuning offer measurable gains, the problem remains far from solved [18].

Despite this progress, most existing benchmarks concentrate on general-purpose tasks, leaving many specialized sectors insufficiently evaluated. In turn, several domain-specific benchmarks have been proposed [67, 32], which demonstrate that LLMs frequently lack the necessary domain expertise and perform poorly without repository-level context, especially when interacting with third-party libraries.

2.2.3. Trends in LLM-Driven Development

The first large-scale conversational interface that introduced LLMs to the general public was ChatGPT². Its release not only attracted unprecedented attention from the research community but also reshaped expectations for how humans interact with such systems. Through natural dialogue, users could query information across diverse domains, request step-by-step solutions to mathematical problems, or even obtain assistance with writing and debugging code [48]. This paradigm of conversation-driven interaction quickly became the default mode of engagement with LLMs, extending naturally to many software engineering tasks.

In software engineering, GitHub Copilot³ illustrates how LLM-based assistance can be embedded into day-to-day workflows, where suggestions for code completion and debugging are continuously generated in the context of the code repository. More recently, interaction paradigms have shifted from question–response or simple interventions toward agentic behavior, where models act autonomously across multiple steps and tools. Agents are typically understood as autonomous systems that can operate over extended horizons, employ diverse tools, and ground their decisions in reasoning and environmental feedback [3]. In software engineering, this shift is visible both in commercial tools, for example, the integrations with JetBrains IDEs⁴, and in research, presenting SWE-agent [62], which autonomously interacts with compilers, debuggers, and version control to resolve complex tasks and achieve higher performance on SWE-bench.

2.2.4. Domain Adaptation and Specialization

Although LLMs are now integrated into popular IDEs, their effective use requires adaptation to constrained settings, including learning enterprise-specific codebases and supporting specialized development tasks. Fine-tuning models on the codebase of a particular project enables them to internalize domain-specific conventions, libraries, and API knowledge, thereby improving performance in downstream tasks [19, 46]. Similarly, models can be specialized to individual programming tasks. A concrete example of this trend is JetBrains' Mellum, a 4-billion-parameter model tailored specifically for code completion and designed for local deployment within repositories [45]. This shows not only the importance of DA but also the need to select models of appropriate size. In practice, we observe a shift toward SLMs for such use cases, due to their efficiency and suitability for targeted tasks [6]. Regardless, fine-tuning a model for domain-specific code generation requires diverse and high-quality training data, which is nontrivial to obtain due to the limited amount of training code available. Moreover, specializing a model not only for DA but also for specific tasks within the domain requires carefully curated datasets [27], which is precisely what we aim to develop.

2.3. Knowledge Transfer in LLMs

To specialize a smaller model for both DA and APR within a repository, we need to explore how domain knowledge can be transferred into LLMs. This section examines the main options and their relevance to our study.

2.3.1. Prompting and Retrieval-Augmented Generation

Prompting allows practitioners to incorporate domain information into LLMs without modifying model parameters. In the context of software engineering, this often takes the form of retrieval-augmented generation (RAG), where the model is provided with additional context such as relevant methods, API documentation, or issue descriptions at inference time [19]. For program repair, RAG can surface similar historical fixes, usage examples of third-party libraries, or failing test cases, effectively turning retrieval into a lightweight substitute for fine-tuning.

However, prompting and RAG alone do not resolve the underlying domain shift between generic pre-training data and the target repository. The model remains constrained by its original parameters and can only leverage contextual hints that fit within the input length.

²<https://openai.com/index/chatgpt/>

³<https://github.com/features/copilot>

⁴<https://www.jetbrains.com/junie/>

2.3.2. Fine-tuning

Many LLMs can already be applied to coding tasks, as they are either trained as general-purpose models, GPT-4 [42], or pre-trained with a particular emphasis on coding such as StarCoder [33], Code Llama [44], CodeGen [40], Mistral [26]. However, when the goal is to adapt an LLM to specialized domains or to refine and improve specific capabilities, fine-tuning is usually employed to adapt the model with task-specific data [23]. In this subsection, we will iterate over the most relevant approaches for teaching or refining models towards a specific objective.

Within the scope of LLMs for software engineering, *full parameter fine-tuning (FFT)* is a standard technique where the model is subject to updating all of its parameters during training which as a result, leads to increased compute costs. However, FFT shows considerable improvements because it manages to fully adapt its parameters to the new tasks which is why it's the leading technique for continued pretraining [7].

To mitigate the high computational costs of FFT, *parameter-efficient fine-tuning (PEFT)* methods have been developed, which update only a small subset of parameters for a target task. Among these, the most widely adopted approach is Low-Rank Adaptation (LoRA) [21]. LoRA freezes the pretrained model weights and instead introduces lightweight trainable matrices, referred to as adapters, into selected weight matrices, typically the attention projection matrices within the Transformer architecture. Building on this idea, QLoRA [13] extends LoRA by loading the frozen base model in 4-bit quantized form while training the adapters in higher precision, thereby achieving substantial memory savings without compromising performance.

2.3.3. Reinforcement Learning

Reinforcement learning (RL) provides an alternative route for training LLMs by optimizing them against task-specific reward signals instead of static supervision. In code generation and program repair, RL frameworks typically use execution results or unit test feedback as rewards, enabling models to iteratively refine candidate programs during training [34, 16]. At the same time, RL introduces substantial engineering overhead: one must define reward functions, maintain interactive environments, and manage the instability of policy optimization. For repository-level APR, this often requires repeatedly building and executing the project under realistic constraints. In this work, we therefore concentrate on SFT, using high-quality trajectories, and treat RL-based approaches as complementary techniques that can further refine a strong base model once efficient and reliable reward signals are available.

2.3.4. Dataset Construction

Applying an optimal fine-tuning strategy by itself is insufficient, since a model's performance is equally shaped by the quality of its fine-tuning dataset. Several aspects are critical when assembling such a dataset: its size, its diversity, and the strategy used to acquire it, whether from pre-existing repositories or via synthetic generation with LLMs. Data can originate from open-source corpora, carefully curated collections aligned with specific research goals, or proprietary industrial datasets, with the latter often offering higher value due to their superior quality and domain specialization [20].

The preprocessing workflow generally involves several core stages. Data extraction first collects the relevant code artifacts from the selected sources. Granularity filtering then adjusts the data to the intended task (e.g., generation at the method level versus the class level). Quality filtering keeps only samples that meet predefined criteria, such as code accompanied by test cases. Deduplication removes repeated examples to mitigate overfitting. In the representation step, the data is converted into suitable formats, such as token sequences or structured representations like Abstract Syntax Trees (ASTs). Finally, data segmentation partitions the corpus into training, validation, and test sets, enabling both the fine-tuning procedure and systematic assessment of model performance [20].

Because specialized or internal repositories are often limited in size, recent work has increasingly turned to synthetic data generated by LLMs. This synthetic data can appear in different forms, including artificial test cases that enhance evaluation robustness or synthetic methods that enlarge the pool of training samples available for downstream models.

The impact of dataset size itself has also been examined, with some studies stating that a model's core knowledge is primarily acquired during pretraining, while fine-tuning serves mainly for alignment [69].

Consequently, data pruning methods have been proposed to address redundancy in training corpora, for example by clustering similar code pairs and discarding low-quality instances. Such approaches can reduce the computational cost of training on large datasets while preserving the model's original performance [52].

In particular, datasets like *OpenCodeInstruct* [2]—currently the largest publicly available code instruction-tuning corpus, with 5 million samples and extensive metadata (unit tests, execution traces, LLM-based judgments)—demonstrate improved results compared with models fine-tuned on smaller datasets. Overall, when fine-tuning LLMs for a specific repository and skill, there is still no clear guideline or “measuring cup” for dataset construction, particularly in scenarios where training examples are scarce.

2.3.5. Knowledge Distillation

One widely adopted approach to constructing synthetic training data is knowledge distillation (KD). As the computational and financial costs of fine-tuning billion-parameter models are high, researchers have turned to more efficient strategies that enable smaller-scale language models, typically containing millions rather than billions of parameters, to inherit the capabilities of their larger counterparts. KD addresses this challenge by training SLMs on datasets produced by teacher models, thereby transferring knowledge without incurring the full cost of large-scale optimization. In doing so, KD offers a practical balance between performance and efficiency: while reducing resource demands, it allows SLMs to approximate the abilities of larger models. This paradigm has demonstrated effectiveness across specialized domains such as mathematics, reasoning, code generation, program repair [55], and agentic tool use.

The typical distillation pipeline begins by steering the teacher model toward a specific skill or domain using carefully designed prompts and instructions. To ensure relevance, seed knowledge (i.e., domain-specific context such as buggy code, test cases, error messages, or task descriptions) is often embedded within the prompt template. For program repair, for example, it is insufficient to simply instruct the teacher to perform code repair, thereby additional context such as a clear task description and feedback should be provided. In this work, we also examine how seed knowledge can be used to elicit not only repaired code but also reasoning traces that make the repair process explicit.

Once the teacher model produces its outputs, the resulting data can be used to supervise the student model through fine-tuning. This teacher-student paradigm, however, is not limited to one-way knowledge transfer. Teacher models can also act as evaluators, providing preferences or corrective feedback during the student's learning process [1, 50]. Such signals are often incorporated through reinforcement learning, allowing the student to refine its behavior [56].

A parallel line of work explores whether models can generate their own training signals through self-reflection [37]. While attractive in principle, its practical benefits are limited. Studies show that, once the cost of repair is considered, independently sampled solutions often match or surpass self-repair, especially under small budgets. Self-repair is most effective when resources are spent on generating diverse initial candidates rather than repeatedly refining a few. Moreover, performance improves substantially only when the feedback itself is artificially enhanced by stronger models or human programmers. These findings suggest that self-refinement alone cannot reliably drive improvement, as its effectiveness depends on the quality of feedback rather than the repair mechanism itself [41].

2.3.6. Reasoning

Learning often arises through cycles of trial, error, and reflection—a process naturally suited to complex problem-solving tasks. While this pattern is intuitive for humans, recent work shows that LLMs can also benefit from explicit reasoning processes [61]. This perspective has motivated research on reasoning-augmented approaches, where models are guided not only to produce outputs but also to verbalize their problem-solving process through structured intermediate steps.

One foundational step toward enabling explicit reasoning in LLMs was Chain-of-Thought (CoT) prompting, which encourages models to externalize intermediate reasoning steps before arriving at a final answer [54]. Although highly effective in domains such as mathematics, applying reasoning to program repair introduces additional challenges: the reasoning must be both reflective (analyzing what went wrong) and actionable (guiding toward a correct fix).

Recent work also highlights that more reasoning is not always better. CoT-style supervision can cause models to overthink, producing excessively long traces that increase latency and token costs without proportional accuracy gains. To address this, Yu et al. [65] propose combining long reasoning traces with shortened, structure-preserving versions, showing that this mixture reduces response length while improving accuracy. Gandhi et al. [15] further demonstrate that models may acquire distinct reasoning behaviors during pre-training, which affect how effectively they learn from feedback during fine-tuning.

These developments establish reasoning as a key mechanism for improving model performance, though its integration into training and inference requires careful design to balance effectiveness with computational cost.

2.4. Related Work

Programming errors are not only common but also costly, and despite advances in automated debugging, producing correct repairs remains a difficult challenge, especially within a specialized code repository [70]. In APR, the objective is to automatically generate patches that correct faulty programs with respect to specifications such as test cases or compiler diagnostics [22]. Traditional approaches have relied on strategies such as search-based patch generation, template-based repair, or constraint solving. With the advent of LLMs, however, repair has increasingly been framed as a learning problem, where models leverage their ability to combine statistical pattern recognition with reasoning over code. Developing models specialized for program repair is inherently challenging. Such models must possess sufficient knowledge of the target codebase, accurately diagnose faults by interpreting error messages, test feedback, or developer guidance, and localize the issue within the relevant code block, ultimately producing a correct and executable patch.

2.4.1. Fine-Tuning Strategies for APR

Classical APR tools generate patches using search-based, constraint-based, or template-based strategies driven by test suites or formal specifications [22, 66]. Learning-based APR replaces hand-crafted patch operators with models that learn bug-fixing patterns from large corpora of patches, often at the granularity of methods or AST edits. These techniques laid the groundwork for LLM-based repair, but they are typically specialized to specific languages and rely on annotated data.

With the emergence of large code models, APR is increasingly framed as a fine-tuning problem: a pretrained model is adapted to generate repairs given buggy code, context, and feedback. Yang et al. [59] provide a recent survey of LLM-based APR systems and categorize them by training objective (FFT versus PEFT), generation regime (single versus multi-turn), and control mechanism (zero-shot versus agentic tool calling). Within this landscape, FFT remains the most straightforward strategy: all weights are updated on repair trajectories, often yielding strong performance but at the cost of high compute and memory requirements.

MORepair [58] goes further by proposing a multi-objective fine-tuning framework that explicitly optimizes both syntactic correctness of patches and the alignment between code changes and their underlying repair rationales [60] using conversational guidance. Their results show substantial gains in improving repair, however it does not place the approach within a specific codebase. Empirical studies show that LLMs often underperform on domain-specific or repository-specific benchmarks, especially when project-specific knowledge is required [67]. Industrial systems increasingly fine-tune smaller models on proprietary repositories to internalize local coding conventions and APIs [19, 45, 6]. Our work follows this direction by adapting an SLM to a single, realistic codebase and analyzing how different fine-tuning methods affect both first-turn and multi-turn repair performance.

2.4.2. Multi-Turn and Agentic Repair with Execution Feedback

Single-shot repair is often insufficient for realistic bugs, where developers iteratively run tests, inspect error messages, and refine patches. Inspired by this process, recent works explore multi-turn and agentic repair frameworks. SWE-agent demonstrates that an LLM equipped with tools such as editors, test runners, and version control systems can autonomously navigate and modify real repositories to solve GitHub issues, achieving strong performance on SWE-bench [62]. Multi-turn APR frameworks such as MORepair already incorporate iterative refinement in their training data [58], while other systems leverage self-reflection and critique to guide successive patch attempts [47, 9].

Execution feedback has also been integrated into the training objective via reinforcement learning. RLTF proposes an online RL framework that uses multi-granularity unit test feedback to refine code LLMs and reports state-of-the-art results on APPS and MBPP [34]. RLEF introduces an end-to-end RL method that grounds LLMs in execution feedback for competitive programming tasks, enabling both small and large models to outperform strong baselines while reducing the number of required samples [16]. Although these approaches primarily target synthesis benchmarks, the underlying idea—optimizing models to leverage feedback over multiple steps—is directly relevant to APR, particularly in agentic settings.

Our work is complementary to these agentic and RL-based frameworks. We focus on strengthening the base repair capability of a small model through supervised trajectories, both with and without explicit reasoning. Such a model can then be embedded as a core component within broader multi-turn or agentic pipelines, benefiting from feedback-aware training without incurring the full complexity of large-scale RL.

2.4.3. Limitations of Current Approaches

Despite rapid progress, several limitations remain in the existing literature.

First, most reasoning-augmented APR systems have been evaluated on generic benchmarks such as HumanEval, MBPP, or competitive programming datasets, where tasks are self-contained and do not require deep repository context [18, 64, 58]. Repository-level studies exist but are less common and often rely on large models whose deployment is costly. There is limited empirical understanding of how well small models, adapted via PEFT, can serve as practical repair assistants in real projects.

Second, reasoning supervision itself introduces trade-offs. Excessively long CoT traces can increase inference cost and sometimes overfit to the task [53]. Moreover, existing work provides limited guidance on how to balance reasoning and non-reasoning examples, how to control reasoning length, and how these factors interact with the main PEFT hyperparameters.

Finally, domain adaptation and data construction remain under-explored for repository-specific APR. Although large instruction-tuning datasets such as OpenCodeInstruct [2] demonstrate the value of scale and rich metadata, they primarily target general-purpose code tasks. There is little research on how much repository-specific APR data is required. This gap motivates our study: we analyze how a single small model can be specialized to a specific repository under realistic data and compute constraints.

3

Methodology

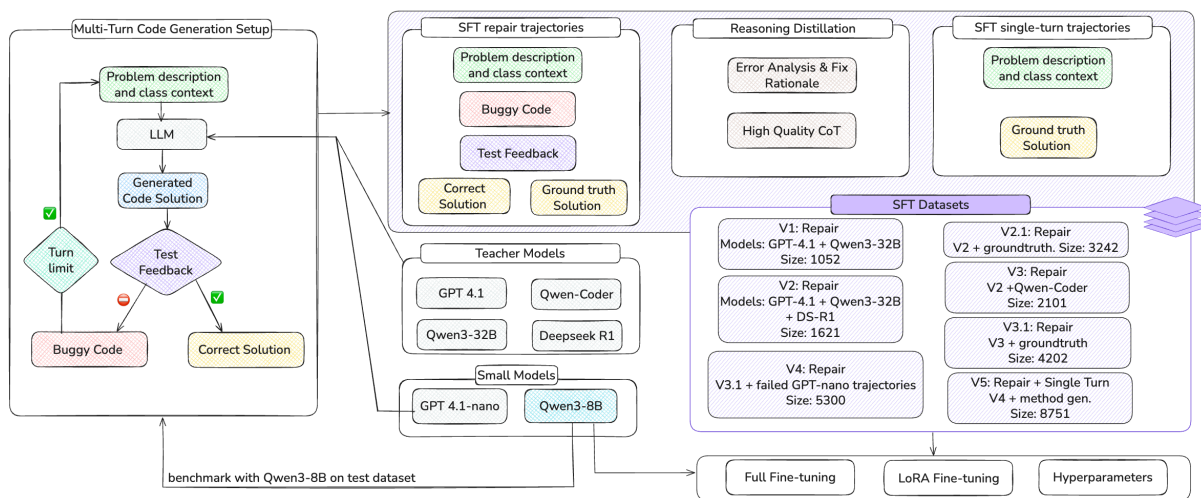


Figure 3.1: Overview of the knowledge distillation pipeline for domain-specific program repair. Teacher models generate multi-turn trajectories on repository methods, which are distilled into training datasets combining repair examples, reasoning traces, and single-turn generation tasks. The student model is then fine-tuned using either LoRA or full fine-tuning under *think* and *no-think* regimes.

Training smaller models for APR presents two fundamental challenges. First, obtaining high-quality training examples from domain-specific codebases is non-trivial, as production repositories rarely provide sufficient diversity of buggy code, feedback, and corrections¹. Second, specialized repositories have considerable complexity through their reliance on internal libraries, APIs, and design conventions, making it improbable that models will generate correct solutions on the first attempt. This stresses the importance of teaching models not merely to synthesize code, but to iteratively repair it through multi-turn refinement. The central question thus becomes: how can a single repository be leveraged to construct repair trajectories that effectively support this learning process?

Our approach to generating high-quality, domain-specific training data leverages a multi-teacher knowledge distillation framework depicted in Figure 3.1. This pipeline is designed to overcome the scarcity of labeled repair data in specialized repositories. It involves:

- **Multi-Turn Code Generation Setup:** An iterative process where an LLM attempts to fix a buggy code snippet. This process incorporates test feedback, simulating a realistic development workflow. The model's generated solutions are evaluated against a test suite, and the feedback guides subsequent repair attempts until a correct solution is found or a turn limit is reached.

¹This scarcity is a byproduct of standard software engineering practices: since the goal is to minimize defects in deployed code, there are fewer real-world buggy examples available for training APR systems.

- **Teacher Models:** The correct solutions are distilled from an ensemble of powerful, general-purpose LLMs. These models act as *teachers*, providing high-quality repair trajectories and reasoning rationales.
- **Reasoning Distillation:** Beyond merely generating correct code, this pipeline allows us to utilize each training trajectory and apply reasoning distillation in a later step. The reasoning blocks obtained include error analysis and fix rationales, which are crucial for training student models to improve their reasoning style.
- **SFT Datasets:** Through iterative refinement, various versions of the SFT dataset are constructed. These datasets incorporate multi-turn repair trajectories, single-turn repair examples, and progressively more complex reasoning traces. The evolution from various versions of the dataset allows for a systematic study of dataset size and composition on model performance and generalization capabilities.

The selection of models for this research was significantly influenced by the accessibility within the JetBrains research department. Given that *Qwen3* models had already been successfully employed in complementary studies, their application in our investigations became a practical requirement. To this end, we implemented a dual SFT strategy for *Qwen3-8B*, our chosen student model, which is inherently designed to operate in two distinct modes: a *thinking* mode, where the model articulates its reasoning within a `<think>...</think>` block prior to emitting the final code, and a *no-thinking* mode, where this reasoning process is suppressed via the `/no_think` control token, leaving the *think* block empty [57]. Throughout this work, we consistently refer to these as the *think* and *no-think* regimes².

This dual configuration not only aligns seamlessly with the model's intrinsic design but is also strategically motivated by practical deployment considerations. While explicit reasoning can undeniably enhance repair quality, it concurrently leads to a substantial increase in token usage and computational latency. On our benchmark, the *thinking* mode took roughly twice as long as the *no-thinking* mode. By training *Qwen3-8B* to reliably handle both formats, we achieve a versatile single student model that can be dynamically switched, through prompt formatting, between a higher-accuracy reasoning mode and a more cost-effective, code-only mode, without the need to maintain separate models for each operational paradigm.

3.1. Repository Selection and Data Extraction

The Django web framework³ was strategically chosen as the target repository for this study, building upon prior investments and expertise within JetBrains Research. Django is a mature, widely-used Python framework with extensive test coverage, well-documented APIs, and established coding conventions. Its codebase contains diverse programming patterns spanning object-relational mapping, authentication, templating, and web protocols, providing a representative sample of real-world software engineering challenges. Furthermore, Django's comprehensive test suite enables reliable verification of generated code, a prerequisite for constructing high-fidelity repair trajectories.

JetBrains researchers had previously curated a dataset from the Django repository, extracting Python methods that satisfy two criteria: they are covered by unit tests, and they belong to classes with clear functional specifications. This extraction process yielded a corpus of 3707 methods across training and validation splits, with 397 methods reserved for the benchmark. The benchmark set was selected to ensure coverage of diverse method complexities and error types representative of the broader codebase. The prompts used during model inference are present in Appendix A, which includes the system prompt (Listing A.1), the base prompt used in the initial turn (Listing A.2), and the prompt used in later turns (Listing A.3) where the model is tasked to repair the buggy code based on unit test feedback.

Each dataset item preserves the following components:

- **File-level context:** The complete source file including all import statements, class definitions, and surrounding methods, but with the target method's implementation removed.
- **Method specification:** The method signature and docstring, which together define the method's interface and intended behavior.

²In both instances, the continuous forms of these regimes, *thinking* and *no-thinking*, are used interchangeably.

³<https://github.com/django/django>

- **Test suite:** A comprehensive set of unit tests that evaluate the method’s functionality.
- **Ground truth implementation:** The original method implementation from the Django repository, preserved for reference and for dataset augmentation strategies described in Section 3.2.

This structured representation ensures that models receive sufficient context to understand the method’s purpose and the codebase conventions, while the test suite provides immediate, automated feedback on the correctness of generated implementations. During both trajectory generation and evaluation, models are presented with the file-level context, the method signature, and the docstring, and are tasked with generating a complete method implementation that satisfies the associated tests.

3.2. Multi-Teacher Knowledge Distillation

Constructing a high-quality repair dataset requires balancing two competing objectives: capturing complex repair patterns that challenge the student model, while simultaneously ensuring sufficient diversity of fault types to prevent overfitting to specific error categories. To achieve this balance, we employed an ensemble of teacher models spanning different capability levels and families.

3.2.1. Teacher Model Selection

We distilled repair trajectories from the following teacher models, each selected for a specific purpose:

- **High-capability models** (*GPT-4.1* [42], *DeepSeek R1* [12], *Qwen3-Coder-480B-A35B-Instruct* [43]): these models are among the leaders on code generation benchmarks and were selected to provide complex, high-quality repair examples that would challenge the student model and demonstrate sophisticated debugging strategies.
- **Mid-tier model**, *Qwen3-32B* [57]: positioned between the strongest teachers and the student model, this model was included to generate a greater proportion of failed attempts, thereby enriching the dataset with diverse buggy instances that reflect realistic intermediate reasoning steps.
- **Small model**, *GPT-4.1-nano* [42]: due to its reduced parameter count, this model produces a wider variety of erroneous outputs, contributing additional fault diversity.

3.2.2. Trajectory Generation

Each teacher model was prompted to perform method generation in a multi-turn setting, where the model receives a task description and class context, generates a method implementation, and then iteratively refines its output in response to test execution feedback. We retained trajectories satisfying one of two conditions:

1. **Successful repair trajectories:** The model’s initial attempt failed unit tests, but a subsequent attempt within the turn limit ($k = 5$) succeeded.
2. **Failed trajectories with ground truth:** The model exhausted all turns without producing a passing solution. For these cases, we paired the final buggy attempt and the test feedback with the ground truth implementation from the Django repository. This pairing is motivated by the observation that final attempts, although incorrect, typically reflect significant progress shaped by test feedback and are structurally closer to the correct solution than earlier attempts.
3. **Successful single turn trajectories:** The model’s initial attempt was successful. We retain the task description and the correct solution that the model can learn to improve the first turn and increase the repair complexity.

To further increase dataset diversity, we expanded each trajectory into two distinct training pairs by associating the same buggy input with (i) the teacher-generated correct solution and (ii) the ground-truth implementation, followed by deduplication. This construction exposes the model to multiple valid repaired methods for identical fault patterns, increasing output variability. Table 3.1 summarizes the incremental construction of the training datasets. Successive versions incorporate additional teacher models or synthesis procedures, allowing controlled analysis of the effects of dataset scale and solution diversity on repair performance.

| Dataset | Size | Teacher Models | Composition |
|---------|--------|---------------------|---|
| v1 | 1,052 | GPT-4.1, Qwen3-32B | Multi-turn repair trajectories |
| v2 | 1,621 | v1 + DeepSeek R1 | Added DeepSeek R1 trajectories |
| v3 | 2,101 | v2 + Qwen3-Coder | Added Qwen-Coder trajectories |
| v3.1 | 4,202 | v3 teachers | v3 + ground truth doubling |
| v4 | 5,300 | v3.1 + GPT-4.1-nano | v3.1 + GPT-4.1-nano failed trajectories |
| v5 | 14,051 | v4 teachers | v4 + single-turn generation |

Table 3.1: Evolution of training datasets through incremental addition of teacher trajectories and synthesis strategies. Each version builds upon previous versions, incorporating new teacher models, ground truth augmentation, and task diversity.

3.3. Reasoning Distillation

Beyond direct repair trajectories, we study whether explicit reasoning traces can further improve the student model’s repair performance. This requires (i) eliciting high-quality reasoning from teacher models and (ii) controlling the length and computational cost of reasoning during both training and inference.

3.3.1. Eliciting Reasoning from Teacher Models

Eliciting reasoning traces from teacher models involves two steps. First, the teacher must be prompted to generate authentic, high-quality reasoning of how the correct solution is derived. We achieve this by providing the buggy code, test feedback, and the ground-truth implementation, and instructing the teacher model to verbalize the debugging process. Second, the resulting traces must be structurally compatible with the student model’s native reasoning format. Imposing unfamiliar or highly structured formats (e.g., rigid markdown templates as opposed to high verbosity) risks disrupting the reasoning patterns learned during pretraining rather than refining them.

These considerations motivate a **minimal-intervention** strategy. We select teacher models from the same architectural family as the student, specifically, *Qwen3-32B* and *Qwen3-235B-A22B-Thinking-2507* [43], and apply lightweight priming that merely opens a reasoning block without imposing a particular structure or requirements.

Before settling on this approach, we explored several alternatives that consistently degraded performance and were therefore not pursued further:

- **Structured markdown reasoning from *GPT-5-mini***⁴ was our first attempt. The model was prompted to generate detailed error analyses and class specifications in a rigid markdown format (see Listing C.5). This approach proved counterproductive: the templated structure diverged sharply from *Qwen3-8B*’s native reasoning style, and fine-tuning on these traces reduced performance below the baseline.
- **Explicit priming** (see Listing A.4 - A.5) was also ineffective. Prompts beginning with dedicated reasoning tags (e.g., `<think>` followed by fixed introductory phrasing such as “*we are given the task to fix ...*”) produced stylistically uniform traces but constrained the model’s natural problem-solving process. This again resulted in a performance drop relative to the baseline.
- **Self-reasoning from *Qwen3-8B*** was attempted by prompting the student model to explain its own repair process given the buggy code and test feedback. The resulting traces were often verbose and stylistically similar to the base model’s default outputs (see Listing C.1). Despite generating a large volume of such traces, including many that exceeded the intended reasoning length, we observed no corresponding improvement in repair accuracy.

Based on these negative results, we adopt the minimal-intervention approach described above. The final training item elicitation prompt that the model learns for repair tasks is shown in Listing 3.1. We define a trajectory to be and we apply this prompt to the training datasets constructed in Section 3.2.

```

1 Please fix the buggy code and reason in a step-by-step way.
2
3 Task description: {task_description}
4 Buggy method: {buggy_code}

```

⁴<https://platform.openai.com/docs/models/gpt-5-mini>

```

5 Test feedback: {test_feedback}
6 Here is the correct solution you will give: {correct_solution}
7
8 <think>\n

```

Listing 3.1: Minimal-intervention reasoning elicitation prompt for repair tasks

3.3.2. Addressing Reasoning Length and Efficiency

Inspection of the distilled traces revealed substantial variation in reasoning length across teacher models (Figure 3.2–3.5). Hence, we also explored whether providing shorter, more concise reasoning examples could yield more efficient supervision. To this end, we applied a **compressed reasoning distillation** approach, prompting *GPT-5-mini* to rewrite each teacher’s reasoning trace into a shorter version while preserving its format, logical structure, and main content (see Listing 3.2). The goal was to produce reasoning that remained useful while avoiding unnecessary verbosity. For example, the reasoning of *Qwen3-32B*, depicted in Listing C.2, was shortened drastically while the format and content were preserved in Listing C.4.

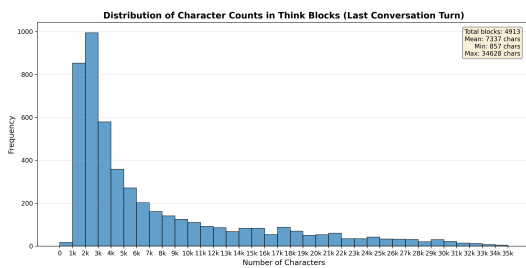


Figure 3.2: The distribution of characters in the reasoning blocks distilled from *Qwen3-8B* for repair trajectories (*v4* dataset).

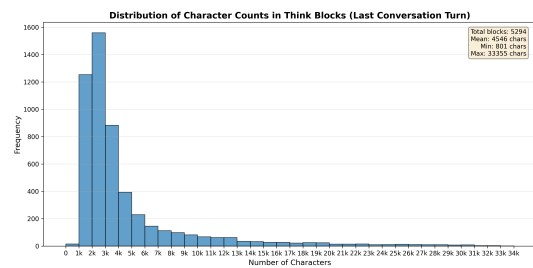


Figure 3.3: The distribution of characters in the reasoning blocks distilled from *Qwen3-32B* for repair trajectories (*v4* dataset).

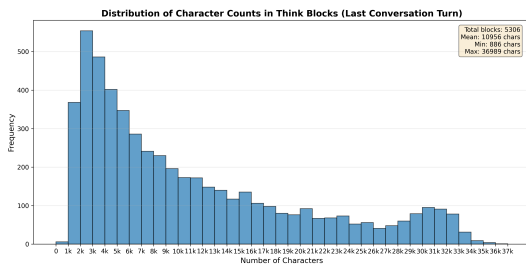


Figure 3.4: The distribution of characters in the reasoning blocks distilled from *Qwen3-235B-A22B-Thinking-2507* for repair trajectories (*v4* dataset).

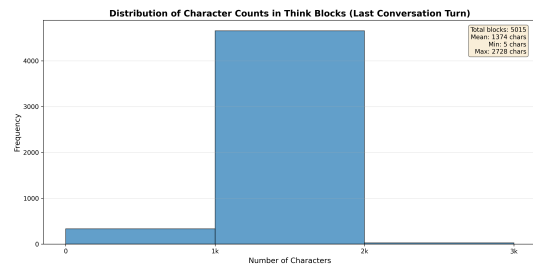


Figure 3.5: The distribution of characters in the reasoning blocks distilled from *Qwen3-32B* and shortened with *GPT-5-mini* for repair trajectories (*v4* dataset).

```

1 Your task is to shorten the following reasoning text to a maximum
2 of 300 tokens while keeping its original logic, tone, and style
3 completely unchanged.
4
5 Rules:
6 1. Do NOT add, remove, or alter any ideas, arguments, or conclusions.
7   However, you are allowed to remove some redundant ideas.
8 2. Do NOT change the reasoning style, tone, or format. Preserve the
9   style of reasoning.
10 3. Only remove unnecessary words or redundant phrasing to reduce
11    length.
12 4. The result must maintain the same reasoning quality and flow.
13 5. Strictly do NOT exceed 300 tokens.
14 6. Output ONLY the modified reasoning. Do NOT include explanations
15    or commentary.
16
17 Previous reasoning:

```

```
18 {previous_reasoning}
```

Listing 3.2: Reasoning compression prompt which was applied to generate shorter reasoning traces. These shorter traces aim to diminish the overthinking problem of the base model.

3.3.3. Single-Turn Reasoning Distillation

In addition to multi-turn repair reasoning, we distill reasoning traces for the single-turn method generation tasks introduced in the v5 dataset (Section 3.2). The elicitation process follows the same minimal-intervention principle but uses a simplified prompt, as no buggy code or test feedback is available (Listing 3.3).

```
1 Task description: {task_description}
2
3 Here is the correct solution you will give: {correct_solution}
4
5 <think>
```

Listing 3.3: Reasoning elicitation prompt for single-turn generation tasks

3.3.4. Hybrid Training with Reasoning and Direct Repair

Finally, we construct a hybrid dataset in which each repair example appears in two forms: a direct repair trajectory and a trajectory augmented with a reasoning trace. The underlying buggy-fixed code pairs are identical, while the only difference is the presence or absence of reasoning. During training, these variants are randomly interleaved, exposing the model to both patterns. This allows the model to learn when to emit a patch directly and when to include reasoning, while keeping supervision on the final solution unchanged.

This setup is loosely inspired by prior work that mixes *thinking* and *no-thinking* supervision to address the overthinking problem in LLMs [65]. We adopt it here as an exploratory mechanism to study how such blending affects reasoning length and repair performance in APR, without assuming any specific benefit.

4

Experimental Setup

Building on the dataset construction and reasoning distillation methods described in Section 3, this chapter specifies the training procedures and evaluation protocols used in our experiments presented in Section 5.

4.0.1. Benchmark

The training and validation corpus consists of 3,707 Django methods with associated test coverage. Each teacher model is applied to all instances using a multi-turn code generation setup. The benchmark set contains an additional 397 methods, each accompanied by unit tests (an average of two tests per method, with a maximum of twenty). Models are evaluated under both *think* and *no-think* prompting modes, using a consistent output format across all training trajectories (for details, see Section 3). All reported results are obtained with a *temperature*=0.6, a maximum of 10000 tokens per generation, and up to $k = 5$ repair turns per benchmark item.

4.0.2. Computational Infrastructure

Our fine-tuning experiments leveraged two H200 Nvidia GPUs¹. For LoRA-based fine-tuning, each GPU was allocated 60-120 GB of RAM and 16 CPU cores. Full fine-tuning experiments, demanding greater resources, utilized 400 GB of RAM. To mitigate GPU memory utilization, we configured DeepSpeed ZeRO-2 offloading², which offloads optimizer states and gradients to the CPU during training. All SFT experiments were conducted using the LLaMA-Factory framework³, a highly configurable platform that supports multiple training methods (including both LoRA and FFT), facilitates efficient hardware utilization, and enables rapid experimentation with minimal boilerplate.

4.1. Supervised Fine-Tuning Strategies

We investigated two supervised fine-tuning regimes: full fine-tuning, which updates all model parameters, and PEFT using LoRA. Both regimes use the same training dataset, differing only in the parameters updated and a small set of method-specific hyperparameters.

Hyperparameters for both SFT methods were selected using the Django validation split, with the goal of stable training and strong functional repair performance under the $Pass@k$ metrics defined in Section 4.1.1. We conducted sanity checks across different hyperparameter configurations to identify reasonable defaults. For LoRA, we experimented with ranks $r \in \{16, 32, 64, 128, 256\}$ and observed diminishing returns and increased overfitting at higher ranks when the dataset was smaller. We therefore adopt $r = 32$ as the default configuration and set the scaling factor to $\alpha = 2r$, which has become a standard practice [7]. We further investigate the impact of LoRA rank on model performance across various dataset sizes in Section 5.2.3.

¹<https://www.nvidia.com/en-us/data-center/h200/>

²<https://www.deepspeed.ai/tutorials/zero-offload/>

³<https://github.com/hiyouga/LLaMA-Factory>

During FFT, higher learning rates (e.g., 1×10^{-4}) caused performance to degrade below the base model’s baseline, suggesting that large parameter updates were overwriting useful pre-trained representations faster than task-specific knowledge could be acquired. We therefore used a learning rate of 1×10^{-5} with 50 warm-up steps and also tested a smaller rate of 5×10^{-6} . On the training dataset, 5×10^{-6} with 100 warm-up steps performed best, while an even smaller rate of 5×10^{-7} led to slightly worse results, likely because the updates were too small for the model to learn within the fixed training budget. All other training settings are shared between LoRA and FFT and presented in Table 4.1. All models are trained for 10 epochs with cosine learning rate schedule to decrease the learning rate smoothly over time, the AdamW optimizer [36], and gradient accumulation such that the effective batch size is 8, which does not stretch our computation resources.

| Parameter | LoRA | FFT | Notes |
|-----------------------------------|--------------------|--------------------|----------------------------------|
| <i>Method-Specific Parameters</i> | | | |
| Rank (r) | 32 | — | Also tested: {16, 64, 128, 256} |
| Alpha (α) | 64 | — | Set to $2r$ |
| LoRA dropout | 0.05 | — | Applied to adapter layers |
| Target modules | All attention | — | Q, K, V, output projections |
| Learning rate | 1×10^{-4} | 5×10^{-6} | FFT also at 1×10^{-5} |
| Warm-up steps | 10 | 100 | Extended for FFT stability |
| <i>Shared Training Parameters</i> | | | |
| Optimizer | AdamW | | $\beta_1 = 0.9, \beta_2 = 0.999$ |
| Weight decay | 0.001 | | |
| LR scheduler | Cosine | | |
| Max gradient norm | 1.0 | | For gradient clipping |
| Gradient accumulation | 8 steps | | Effective batch size: 8 |
| Batch size (per device) | 1 | | Train and eval |
| Epochs | 10 | | |
| Precision | bfloat16 | | |
| Max sequence length | 32,768 tokens | | |

Table 4.1: Hyperparameter configurations for LoRA and full fine-tuning. Shared parameters apply to both methods.

Training data is provided in the ShareGPT JSON format ⁴, where each repair instance is represented as a short multi-turn dialogue containing the task description, the buggy method, the test feedback, and the corrected solution. We define a **trajectory** to contain one training item in the respective format. We also experiment with training on method generation trajectories, where the training item contains the task description and the correct solution. LlamaFactory then adapts this format for the specified model’s template ⁵. An example instance is shown in Listing 4.1 and the system prompt can be found in Listing A.1.

```

1 [
2   {
3     "system": "system prompt",
4     "conversations": [
5       {
6         "from": "user",
7         "value": "task description"
8       },
9       {
10        "from": "AI",
11        "value": "buggy method"
12      },
13      {
14        "from": "user",
15        "value": "test feedback"
16      },
17      {

```

⁴<https://github.com/hiyouga/LlamaFactory/blob/main/data/README.md#sharegpt-format>

⁵<https://github.com/hiyouga/LLaMA-Factory/tree/main?tab=readme-ov-file#supported-models>

```

18     "from": "AI",
19     "value": "correct solution"
20   }
21 ]
22 }
23 ]

```

Listing 4.1: The ShareGPT-formatted training instance of LlamaFactory that we used to format our training trajectories. This format is compatible with the chat instruction following format of Qwen3 models.

4.1.1. Functional Correctness Metrics

Our primary measure of functional repair performance is $Pass@k$ introduced by Chen et al., [10], which reports the proportion of benchmark methods for which at least one of the top- k generated candidates passes all tests. We compute $Pass@k$ for $k \in \{1, 2, 3, 4, 5\}$, capturing both first-try success and the benefits of additional repair turns.

We additionally report $ExactMatch(EM)$, defined here as the fraction of benchmark items where the generated method exactly matches the reference implementation at the text level. While EM is stricter than test-based metrics and is fragile to semantically equivalent but syntactically different repairs, it provides a useful upper bound on how often the model reproduces the original code.

Finally, we use $CodeBLEU$ ⁶ to assess structural and semantic similarity between the generated method and the reference. $CodeBLEU$ combines n-gram overlap, abstract syntax tree similarity, and data-flow consistency, offering a more code-aware signal than EM for cases where multiple correct implementations exist.

4.1.2. Reasoning Quality Assessment

For models trained on reasoning-augmented datasets, we also measure how closely the student reasoning aligns with the teacher’s. We focus on textual similarity and length statistics:

- $ROUGE-L$ ⁷: computes the longest common subsequence between teacher and student reasoning traces, providing a sequence-level overlap score that is sensitive to token order but tolerant to gaps. We use it to capture whether the student follows the teacher’s overall reasoning trajectory.
- $JaccardSimilarity(4\text{-grams})$ ⁸: measures the intersection over union of 4-gram sets extracted from the two traces, indicating if there is a reuse of phrases or domain-specific terminology. We choose a 4-gram Jaccard score to complement $ROUGE-L$ with a stricter view on lexical overlap.
- **Reasoning length statistics:** track the average number of words and tokens in the generated `<think>` blocks. These statistics let us quantify whether SFT leads to shorter, more concise explanations or encourages long reasoning.

4.1.3. Baseline and Generalization Assessment

In our main experiments, we analyze the impact of dataset size and key hyperparameters on fine-tuned *Qwen3-8B* checkpoints, comparing them against the base model. To validate the generalizability of our approach, we also evaluate it on additional models using a subset of the training datasets:

- **Larger Qwen3 models:** *Qwen3-14B* and *Qwen3-32B*, which can indicate whether our method improves stronger models.
- **Other open-weight models:** *Llama-3.1-8B-Instruct* and *Mistral-8B-Instruct-2410*, fine-tuned on the same APR data to test whether our findings transfer across other families of models.

To understand whether the models overfit to the specific repair task format, we additionally evaluate selected checkpoints on SWE-Agent [62], which scaffolds SWE-bench [28] in a tool-calling, agentic setup. We use the Django subset of SWE-Agent, which contains 220 GitHub issues from the Django repository. This evaluation allows us to assess whether fine-tuning on our Django repair dataset degrades the model’s ability to perform repair in an agentic, multi-step interaction format, and whether repository-specific training on Django improves issue resolution on the same repository.

⁶<https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/code-to-code-trans/evaluator/CodeBLEU>

⁷<https://aclanthology.org/W04-1013/>

⁸https://scikit-learn.org/stable/modules/generated/sklearn.metrics.jaccard_score.html

To account for natural score fluctuations due to the temperature setting during sampling, we evaluated the base model (*Qwen3-8B*) five times on the Django benchmark. Figures 4.1 and 4.2 show that the standard deviation across trials is approximately ± 0.4 – 0.9% and we admit these fluctuations in the results presented in Section 5.

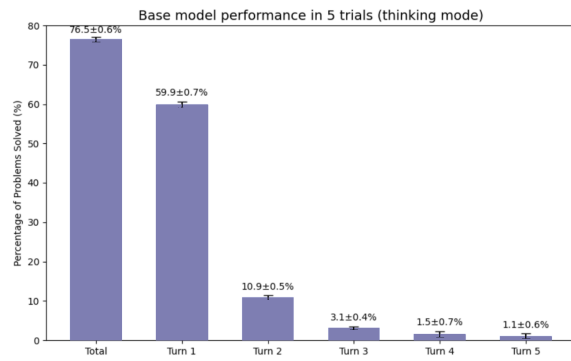


Figure 4.1: Pass@k performance of *Qwen3-8B* on Django across five evaluation runs in *thinking* mode.

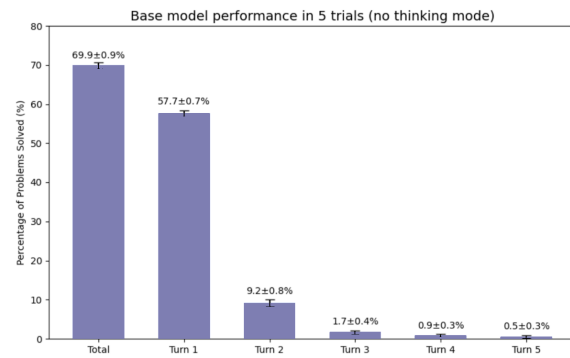


Figure 4.2: Pass@k performance of *Qwen3-8B* on Django across five evaluation runs in *no-think* mode.

5

Results

This chapter reports the empirical evaluation of the multi-teacher distillation pipeline introduced in Section 3. We establish baseline performance across model families and inference regimes (Section 5.1), then analyse supervised fine-tuning without explicit reasoning, focusing on dataset composition, scale, and fine-tuning strategy (Section 5.2). We next evaluate reasoning-augmented fine-tuning, examining its effects on repair accuracy, reasoning stability, teacher alignment, and token efficiency (Section 5.3), before analysing generalisation across model families and sizes (Section 5.5).

5.1. Baseline Performance

Table 5.1 summarizes $Pass@k$ results across base models before fine-tuning. Several patterns emerge from these results. Within the *Qwen3* family, performance scales consistently with model size, and each model demonstrates a clear performance gap between *think* and *no-think* modes. This separation confirms the value of internal reasoning traces and justifies treating both modes explicitly in our fine-tuning pipeline. Among open-source models of comparable size, *Qwen3-8B* (our baseline model) consistently outperforms both *Llama* and *Minstral* variants. Proprietary models establish the performance ceiling, with *GPT-4.1* setting the upper bound on this benchmark.

| Model | @1 | @2 | @3 | @4 | @5 | Total | Mode |
|---------------------------|------|------|-----|-----|-----|-------|---------|
| <i>Qwen2 Family</i> | | | | | | | |
| Qwen3-8B | 59.9 | 10.9 | 3.1 | 1.5 | 1.1 | 76.5 | Think |
| Qwen3-8B | 57.7 | 9.2 | 1.7 | 0.9 | 0.5 | 69.9 | Nothink |
| Qwen3-14B | 63.7 | 11.8 | 3.3 | 1.5 | 1.0 | 81.4 | Think |
| Qwen3-14B | 62.0 | 10.3 | 1.8 | 1.5 | 0.3 | 75.8 | Nothink |
| Qwen3-32B | 64.9 | 12.8 | 5.0 | 1.0 | 0.7 | 84.5 | Think |
| Qwen3-32B | 62.0 | 9.0 | 3.2 | 2.5 | 0.0 | 77.7 | Nothink |
| <i>Other Open Models</i> | | | | | | | |
| Llama-3.1-8B-Instruct | 41.1 | 11.3 | 5.8 | 4.0 | 0.8 | 63.0 | Think |
| Mistral-8B-Instruct-2410 | 41.6 | 12.8 | 2.5 | 1.8 | 1.0 | 59.7 | Think |
| <i>Proprietary Models</i> | | | | | | | |
| GPT-4.1 | 74.5 | 9.5 | 2.6 | 1.4 | 0.5 | 88.5 | Think |
| GPT-4.1-nano | 58.0 | 6.7 | 1.2 | 0.6 | 0.5 | 67.0 | Think |

Table 5.1: Baseline results on the Django benchmark in *think* and *no-think* modes. The table shows that model performance scales with size and that reasoning consistently improves accuracy, showing the performance gap that our fine-tuning methods aim to reduce.

Three observations from these baseline results shape the remainder of this analysis. First, the strongest open-source model, *Qwen3-32B*, falls only a few points below *GPT-4.1* despite having approximately

56× fewer parameters, suggesting that the benchmark is approaching saturation. Consequently, fine-tuning gains are expected to be modest. Second, larger models solve more problems on the first attempt, while smaller models rely more on iterative repair across subsequent turns. Third, most successful repairs occur in the first two turns, with diminishing improvements in later turns. These patterns validate our approach of fine-tuning for the repair task, though the benchmark may not capture all benefits comprehensively.

5.2. Supervised Fine-Tuning Without Reasoning

In this section, we present results from fine-tuning experiments that explore the impact of dataset size, compare LoRA and FFT strategies, and examine the main hyperparameters affecting training stability.

5.2.1. Dataset Evolution and Multi-Teacher Distillation

Table 5.2 tracks performance across incremental dataset versions, comparing LoRA and FFT under both evaluation modes. Three key findings emerge from this progression. First, increasing dataset size consistently improves performance, with $Pass@5$ rising from 77.6% ($v1$, LoRA, think) to 82.4% ($v4$, LoRA, think). Examining fine-grained scores across epochs shows modest repair improvement with the initial $v1$ dataset (Figure B.1). Doubling the dataset by adding items distilled from two teacher models ($v3$ dataset) substantially boosts repair performance (Figure B.2), though the *no-think* regime remains somewhat unstable. Only when we introduce diverse training items, doubling the dataset with ground-truth solutions and adding failed trajectories from a smaller model (*GPT4.1-nano*), does the model perform steadily across turns in the *no-think* regime (Figures 5.1 and 5.2). Second, the transition from $v4$ to $v5$, which adds 8751 single-turn generation examples, yields substantial *no-think* gains (+1.8% for LoRA, +2.0% for FFT) and improves performance over the base model overall. Third, the performance gap between *think* and *no-think* modes narrows through both LoRA and FFT, indicating that our approach successfully transfers reasoning capabilities to the non-reasoning regime.

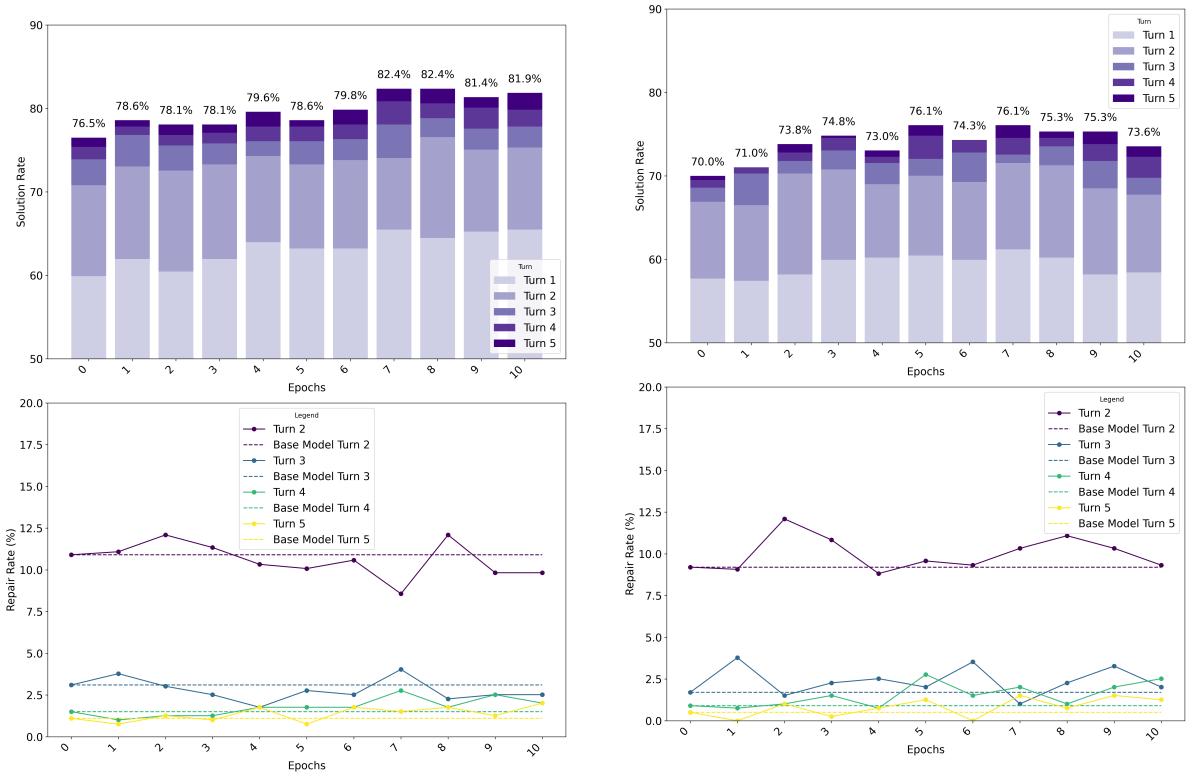
| Dataset (size) | LoRA ($rank = 32$) | | FFT | |
|----------------|----------------------|---------------------|--------------|---------------------|
| | Think | Nothink | Think | Nothink |
| $v1$ (1,052) | 77.6% (+1.1) | 73.0% (+3.1) | 71.5% (-5%) | 64.0% (-5.9%) |
| $v2$ (1,621) | 79.1% (+2.6) | 75.3% (+5.4) | — | — |
| $v3$ (2,101) | 79.8% (+3.3) | 74.8% (+4.9) | 79.8% (+3.3) | 74.3% (+4.4) |
| $v3.1$ (4,202) | 80.4% (+3.9) | 75.6% (+5.7) | 79.1% (+2.6) | 74.8% (+4.9) |
| $v4$ (5,300) | 82.4% (+5.9) | 76.3% (+6.4) | 80.8% (+4.3) | 77.6% (+7.7) |
| $v5$ (14,051) | 80.9% (+4.4) | 78.1% (+8.2) | 79.6% (+3.1) | 79.6% (+9.7) |

Table 5.2: Performance evolution across dataset versions for *Qwen3-8B*. Dataset size increases from 1052 ($v1$) to 14051 ($v5$) examples. LoRA achieves peak *think* mode performance with $v4$ (82.4%, +5.9 points), while FFT peaks in *no-think* mode with $v5$ (79.6%, +9.7 points), nearly closing the *think-no-think* gap.

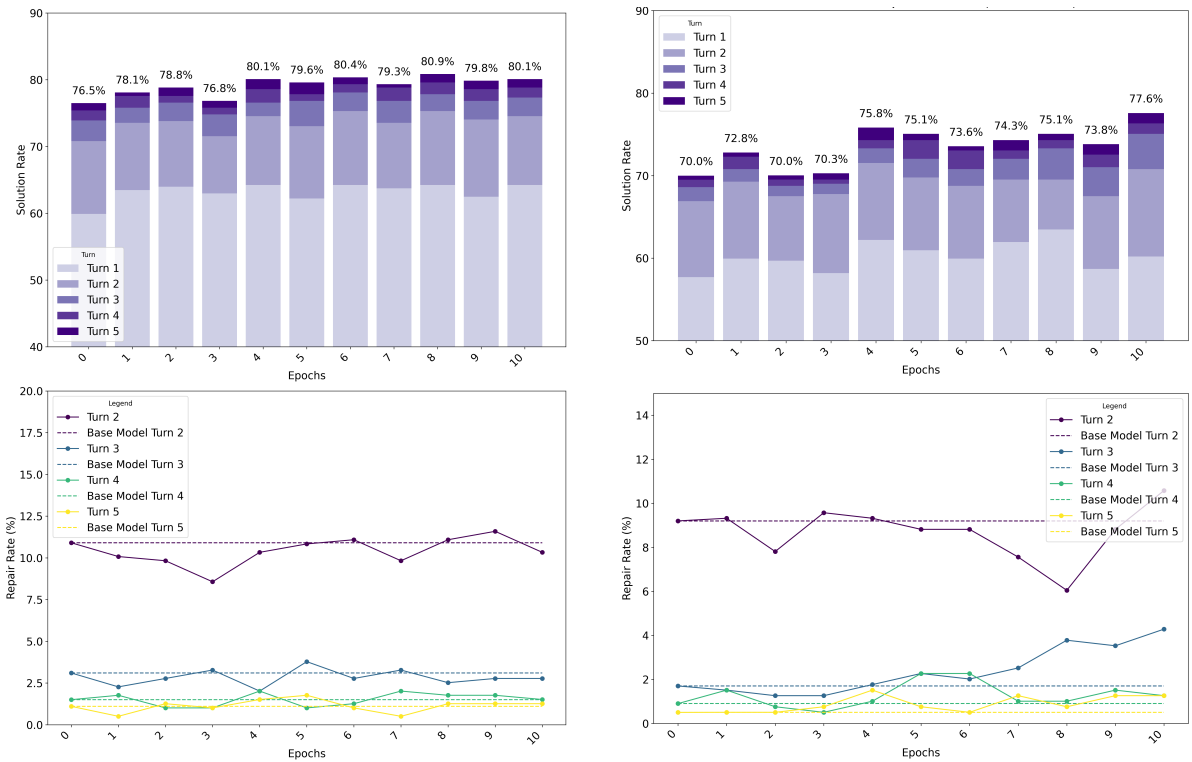
5.2.2. LoRA versus Full Fine-Tuning

Comparing fine-tuning strategies in Table 5.2 reveals complementary strengths. LoRA achieves the highest performance in think mode (82.4% on $v4$), while FFT scores high in *no-think* mode (79.6% on $v5$) with significant improvement in repair over baseline. Moreover, FFT achieves identical performance in both modes (79.6%), effectively closing the gap between reasoning and non-reasoning regimes. In contrast, LoRA maintains a 2.8% gap, suggesting partial reliance on reasoning even when trained without explicit reasoning traces.

However, the benefits of FFT only become apparent with larger and more diverse datasets, where its performance becomes stable and competitive. With smaller datasets, FFT often underperforms, as seen in Figure B.2. Performance stabilizes with more training examples (Figure B.3) and shows clear improvements on our largest training dataset (Figure 5.2). These results suggest that the optimal method depends on the use case. For smaller training datasets, LoRA provides sufficient capacity for effective learning. For larger datasets, particularly when combining multiple tasks, FFT better retains diverse training examples. We further explore whether increasing LoRA rank could match or exceed FFT performance in Section 5.2.3.

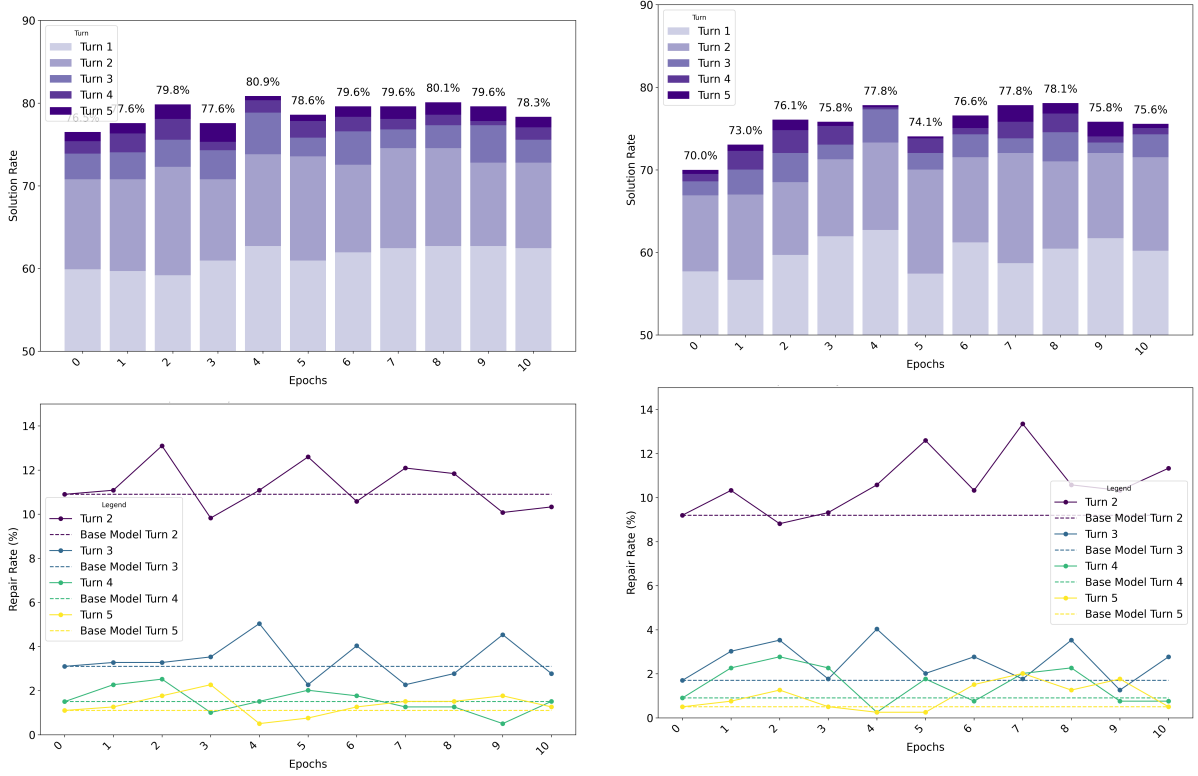


(a) LoRA fine-tuning on the v4 repair-only dataset, evaluated in *thinking* (left) and *no-thinking* (right) regimes. Performance improves in both regimes, with stronger gains in repair turns under *no-thinking* inference.

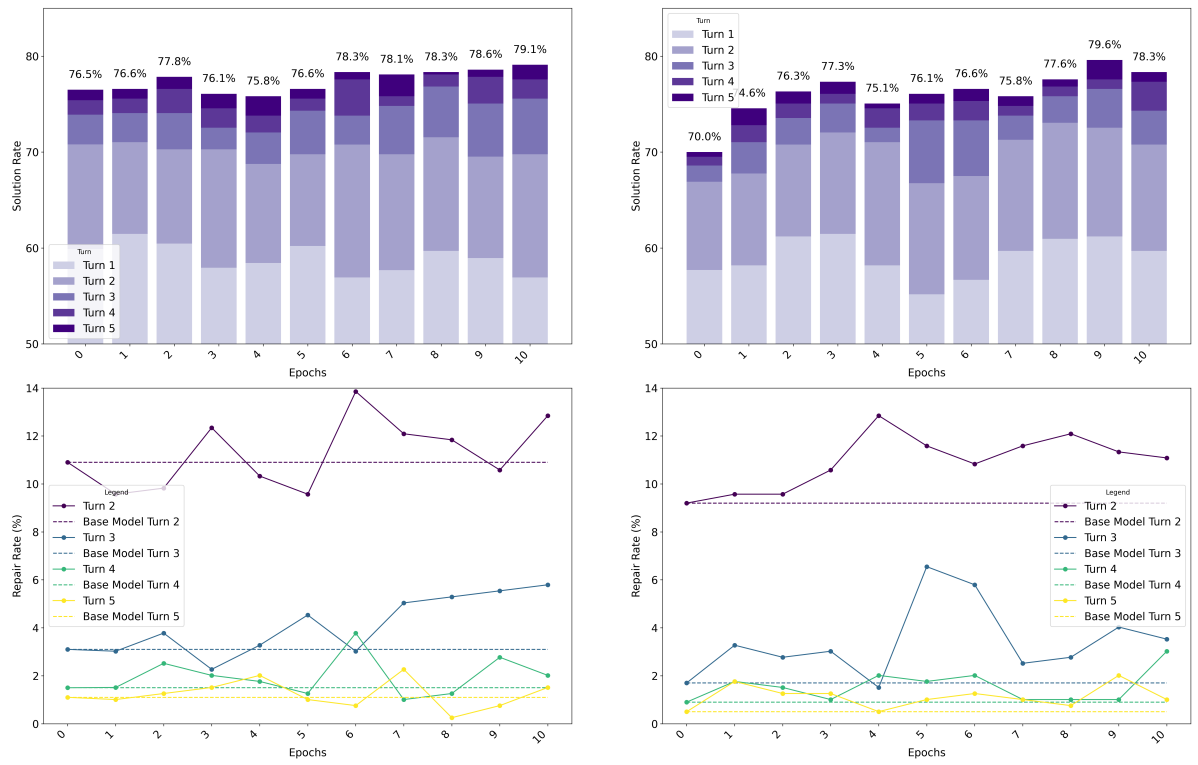


(b) FFT on the v4 repair-only dataset, evaluated in *thinking* (left) and *no-thinking* (right) regimes. Gains in *no-thinking* mode primarily emerge in later repair turns.

Figure 5.1: Training dynamics of LoRA and FFT on the v4 dataset. Bars show single-turn accuracy at each checkpoint, while lines show repair performance across turns on the Django benchmark. Both approaches improve overall accuracy, with different performance and repair patterns across inference regimes.



(a) LoRA fine-tuning on the v5 dataset (repair and single-turn generation), evaluated in *thinking* (left) and *no-thinking* (right) regimes. Including single-turn examples improves repair performance compared to repair-only training.



(b) FFT on the v5 dataset, evaluated in *thinking* (left) and *no-thinking* (right) regimes. Performance in the *no-thinking* regime approaches that of *thinking* inference.

Figure 5.2: Training dynamics of LoRA and FFT on the v5 dataset, which includes both method generation and repair trajectories. Bars show single-turn accuracy at each checkpoint, and lines show cumulative repair performance across turns on the Django benchmark.

Figure 5.1 illustrates training dynamics across epochs for both methods on the *v4* dataset. LoRA demonstrates steady improvement throughout training, with performance plateauing between epochs 7-10. FFT initially lags but shows rapid convergence in later epochs, particularly in *no-think* mode. Additionally, LoRA exhibits more stable repair rates across turns 2-5, maintaining consistent improvement over the base model, while FFT shows greater variance with occasional drops in intermediate turns before recovering in the final turn.

An important observation from these experiments is that our dataset generation approach, which excluded reasoning traces from repair examples, did not compromise the structural or semantic aspects of the base model’s reasoning capability. We observe improvements in both modes, with the model evaluated in *think* regime maintaining its advantage. These results suggest that the diversity and quality of repair examples expose the model to repository patterns (evidenced by single-turn improvements), though the 82% resolution rate may indicate missing context that could further improve performance when provided.

In our experimental setup, LoRA emerges as the preferred method because it enables the model to learn both repair and single-turn generation, achieves the highest benchmark scores in both modes, and requires significantly less time and computational resources - LoRA training was approximately four times faster when trained on two H200 Nvidia GPUs.

5.2.3. Hyperparameter Analysis: LoRA Rank Selection

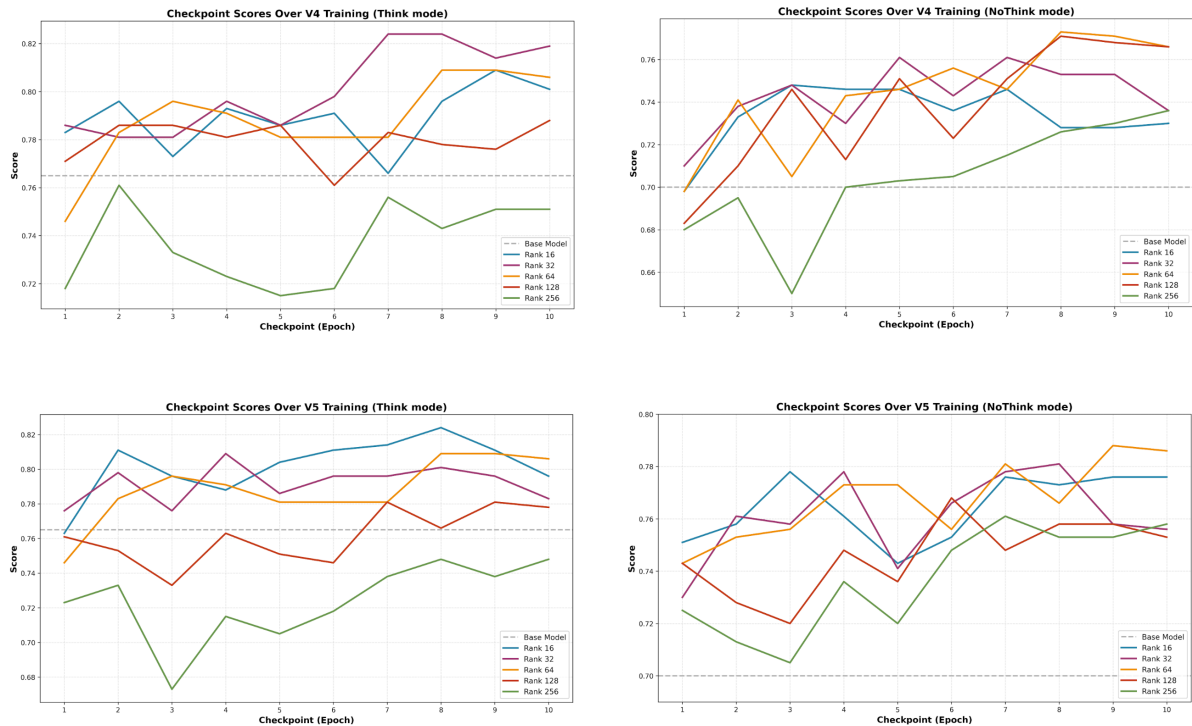


Figure 5.3: Impact of LoRA rank on training stability and final performance across *v4* (repair-only, top row) and *v5* (repair with method generation, bottom row) datasets. Lower ranks (16–32) converge reliably within 10 epochs. Mid-range rank (64) provides stable nothink performance. Higher ranks (128–256) consistently underperform due to overfitting.

We evaluate adapter capacity by varying LoRA rank, $r \in \{16, 32, 64, 128, 256\}$, on *v4* (repair-only) and *v5* (repair with method generation) datasets. Figure 5.3 presents $Pass@5$ performance across all 10 training checkpoints, revealing both convergence behavior and rank stability.

On *v4*, rank 32 exhibits the strongest performance, climbing steadily to 82.4% before plateauing at epoch 7. In nothink mode, rank 64 demonstrates better stability, maintaining consistent improvement to 77.3%, while rank 32 plateaus earlier at 76.1%. Ranks 16 and 64 follow similar patterns in *think* mode

but remain 1–2% below rank 32. Higher ranks, 128 and 256, show unstable behavior and consistently underperform.

On *v5*, rank 16 produces the smoothest curve in *think* mode, reaching 82.4% with minimal fluctuation after epoch 5. Ranks 32 and 64 remain competitive but show slightly more variance. However, in *no-think* mode, rank 16 becomes unstable, dropping sharply at epoch 5, while rank 64 maintains the most consistent trajectory to 78.8%.

Across all configurations, the choice of LoRA rank has a direct impact on training behaviour. Ranks above 64 did not yield performance gains and were consistently associated with noisier or less stable learning curves, whereas ranks between 16 and 64 converged reliably. Learning curves typically stabilize around epoch 8, suggesting that the model requires multiple passes over the dataset to fully learn the repair patterns. Overall, these results support existing research, which stresses the importance of choosing the right hyperparameters for a concrete setup. For fine-tuning on a particular codebase, we expect training data to be relatively moderate in size, making LoRA, with rank of 32 or 64, effective for both code memorization and format learning. We suggest that FFT could be applied to tasks that require higher memorization, such as class-level generation as an initial step before specific tasks (e.g., repair, code completion) are learned.

5.3. Reasoning-Augmented Fine-Tuning

Table 5.3 compares reasoning-only and mixed training configurations (combining reasoning and non-reasoning examples) with our best non-reasoning APR SFT model (*v4*, LoRA $r = 32$). Training exclusively on reasoning trajectories does not improve repair performance compared to our non-reasoning experiments: all reasoning-only models fall short of 82.4% *Pass@5* in *think* mode and achieve no more than 76.3% in *no-think* mode, regardless of teacher model size (8B, 32B, or 235B) or fine-tuning method.

| Training Configuration | Rank | Think | Nothink |
|---|------|--------------|--------------|
| <i>Baseline (No Reasoning)</i> | | | |
| Qwen3-8B trained on <i>v4</i> dataset (best) | 32 | 82.4% | 76.3% |
| <i>Reasoning Only</i> | | | |
| Qwen3-8B-reasoning-32B-teacher- <i>v4</i> | 32 | 78.8% | 75.3% |
| Qwen3-8B-reasoning-32B-teacher- <i>v4</i> -short | 64 | 77.1% | 73.3% |
| Qwen3-8B-reasoning-235B-teacher- <i>v4</i> | 32 | 77.6% | 75.8% |
| Qwen3-8B-reasoning-32B-teacher- <i>v5</i> | 32 | 78.8% | 75.8% |
| Qwen3-8B-reasoning-32B-teacher- <i>v5</i> | FFT | 79.6% | 75.3% |
| <i>Hybrid (Reasoning + Nothink)</i> | | | |
| Qwen3-8B-reasoning-32B-teacher- <i>v4</i> -with-non-reasoning | 32 | 79.8% | 76.6% |
| Qwen3-8B-reasoning-32B-teacher- <i>v4</i> -with-non-reasoning | 64 | 80.1% | 76.3% |
| Qwen3-8B-reasoning-32B-teacher- <i>v4</i> -short-with-non-reasoning | 64 | 77.8% | 78.3% |
| Qwen3-8B-reasoning-32B-teacher- <i>v5</i> -with-non-reasoning | 64 | 79.8% | 79.6% |
| Qwen3-8B-reasoning-235B-teacher- <i>v5</i> -with-non-reasoning | 64 | 78.8% | 78.6% |

Table 5.3: Performance of reasoning-augmented models compared to the best non-reasoning baseline (*v4*, LoRA $r = 32$). Hybrid models combine reasoning and nothink examples. The experiment name is composed of <base model>-reasoning-<teacher size>-teacher-<dataset>-<specification>.

Hybrid configurations, which mix reasoning and non-reasoning trajectories, recover most of the non-reasoning performance and modestly improve *no-think* scores. The best mixed model, *Qwen3-8B* trained with LoRA rank 64 on a combination of reasoning trajectories distilled from *Qwen3-32B* and non-reasoning repair examples from *v5*, reaches 79.8% in *think* mode and 79.6% in *no-think* mode. This performance is comparable to our best non-reasoning model trained on *v5* with FFT (79.6% nothink).

Overall, explicit reasoning supervision does not increase *Pass@5* beyond strong non-reasoning APR SFT. Its main benefit is enabling shorter and cheaper reasoning traces (Section 4.1.2), while hybrid training largely matches, rather than exceeds, the best non-reasoning configurations.

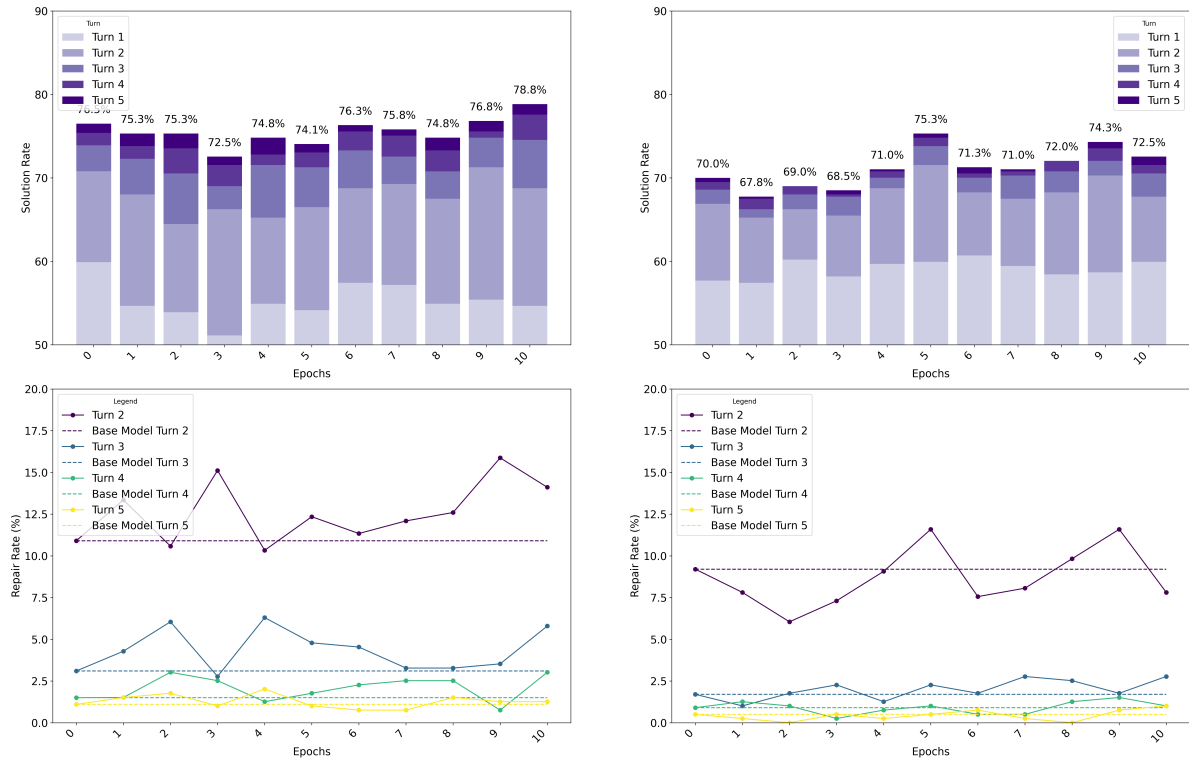


Figure 5.4: LoRA fine-tuning with repair items (v4) and with added distilled reasoning from *Qwen3-32B* in *thinking* (left) vs *no-thinking* (right) mode. In *thinking* mode, we notice unstable performance in method generation, while the *no-think* regime was not affected.

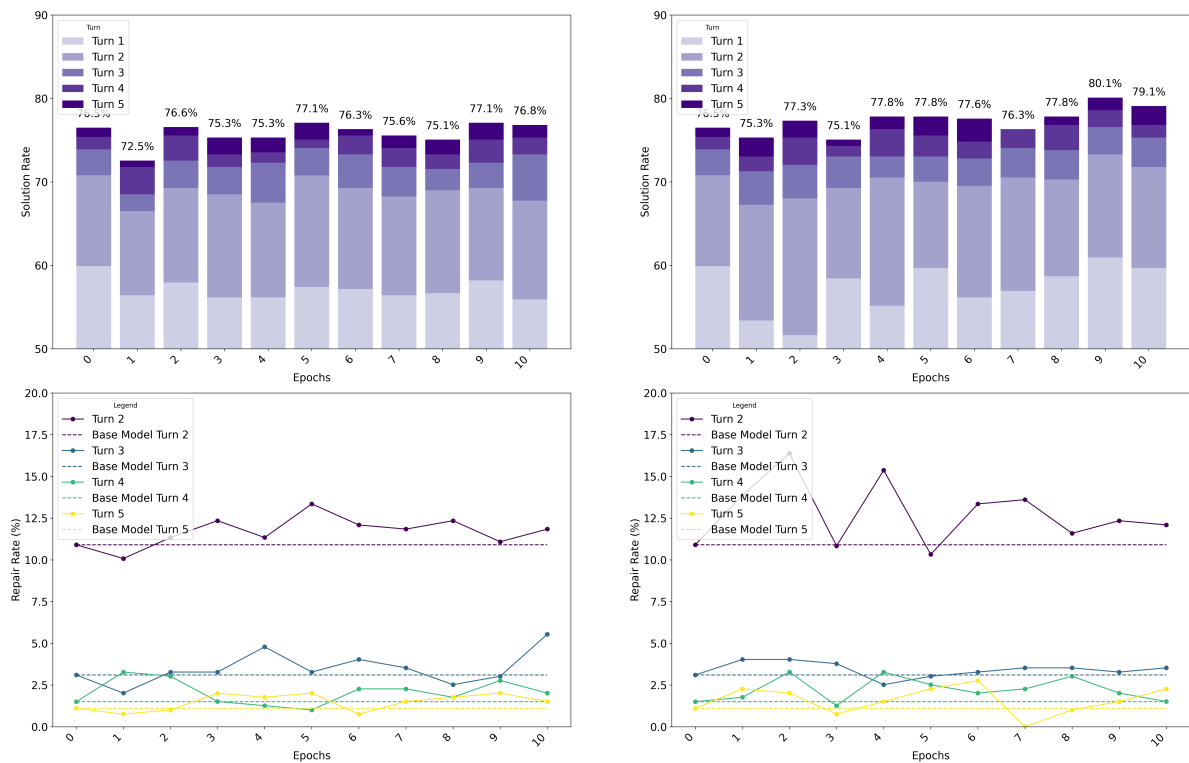
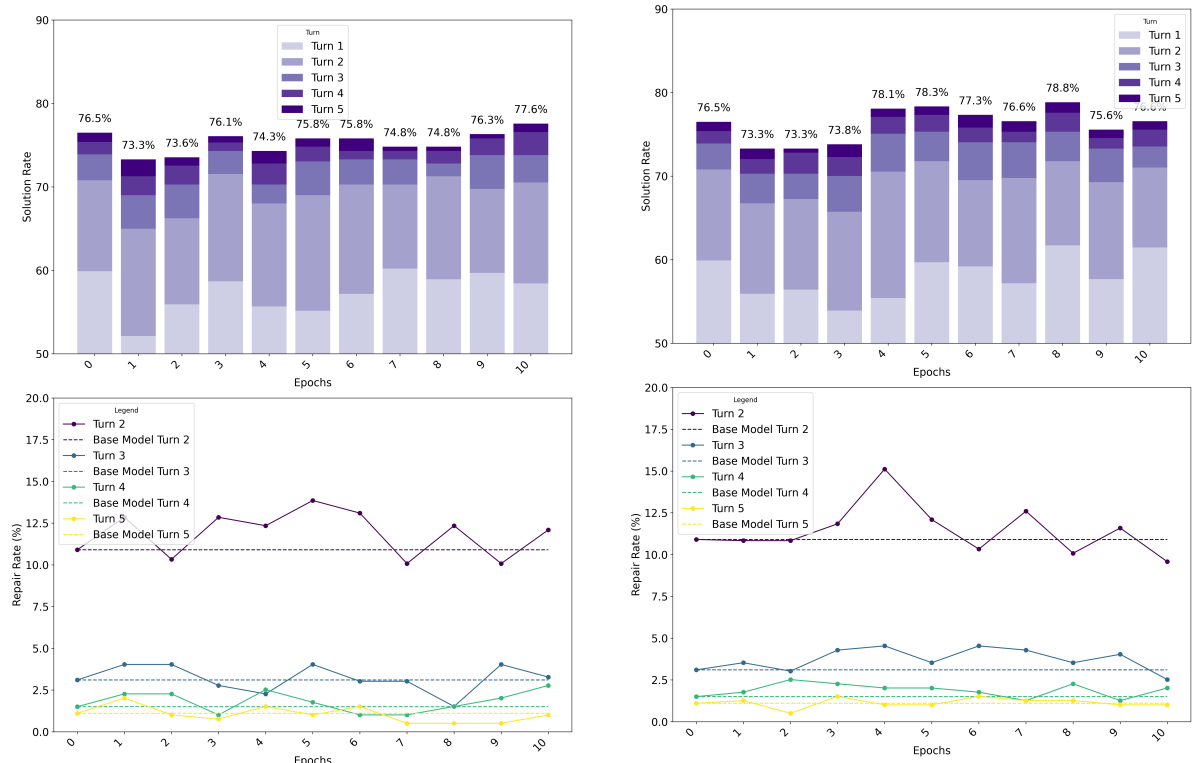


Figure 5.5: Right: LoRA SFT with repair items (v4 dataset) and with added **shortened** distilled reasoning from *Qwen3-32B*. Left: LoRA SFT with repair items and method generation (v5 dataset) with added distilled reasoning from *Qwen3-32B*.

Examining score dynamics across checkpoints trained with reasoning-augmented repair trajectories distilled from *Qwen3-32B* (Figures 5.4), we observe that method generation is negatively impacted compared to training without reasoning. Even when we intervene with shorter reasoning or blend reasoning repair trajectories with non-reasoning trajectories (Figure 5.5) to introduce reasoning gradually, two issues emerge that slightly destabilize the model.

First, the training signal appears to emphasize reasoning learning over learning the corrected code, likely because reasoning tokens are substantially longer than the target method. Second, reasoning that differs in tone, structure, or format from the base model can affect learning. For example, when reasoning is elicited from distillation using priming (see Section 3.3.1), benchmark performance either stagnates or declines (Figure 5.7), even when the reasoning content is correct and aligns with the model’s natural reasoning. This fragility to subtle or dramatic changes in reasoning style may disproportionately affect smaller models, we hypothesize that larger models are more robust, though this remains untested.



(a) LoRA SFT on repair dataset (v4 dataset) augmented with distilled reasoning from *Qwen3-235B-A22B-Thinking-2507* and evaluated in the thinking regime.

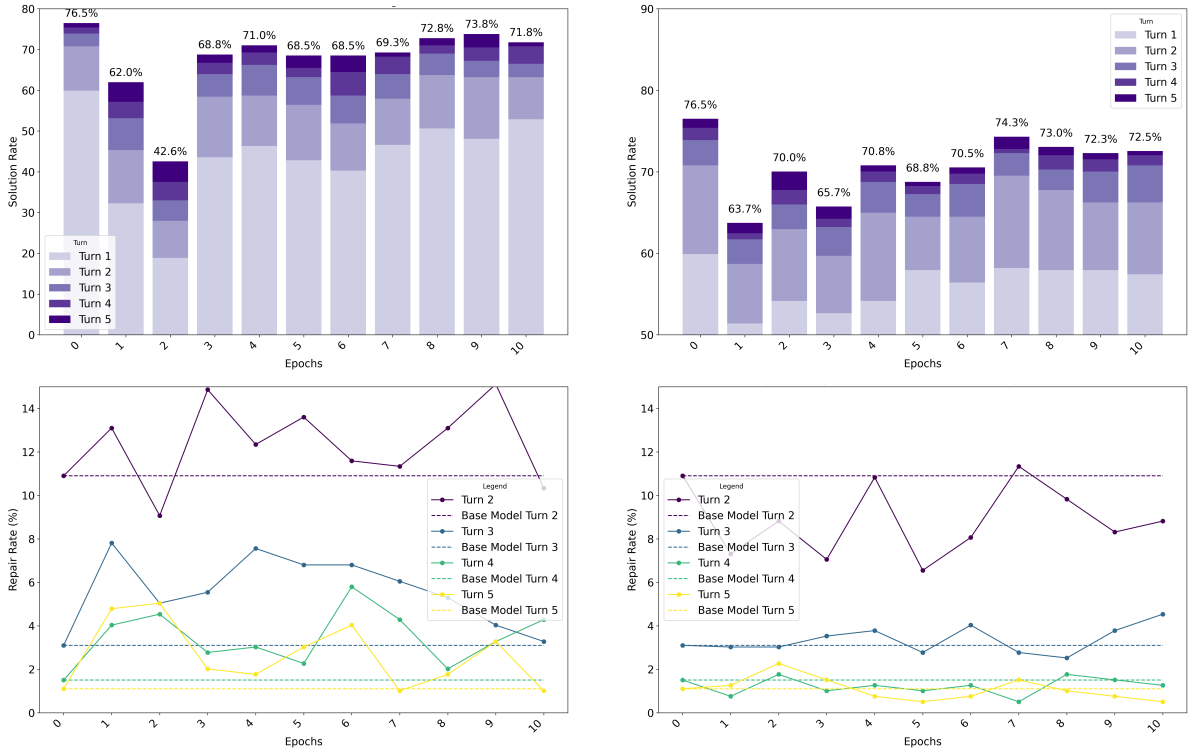
(b) LoRA SFT on repair dataset and method generation (v5 dataset) augmented with distilled reasoning from *Qwen3-235B-A22B-Thinking-2507* and evaluated in the thinking regime.

Figure 5.6: Benchmark performance across checkpoints after LoRA SFT with distilled reasoning from *Qwen3-235B-A22B-Thinking-2507* and trained on *Qwen3-8B* base model. The bar plots represent the score of the model at the respective training epoch, while the line plots represent the score obtained in subsequent turns where the model had to repair. Training with reasoning distilled from a significantly larger teacher model resulted in instability of the model during method generation. Adding slightly diverse reasoning: method generation and repair (right figure), allowed the model to improve slightly against the baseline.

Another key observation is that the teacher model’s family and the reasoning structure strongly influence SFT outcomes. Using a teacher model closer in family and size to the base model yields more stable and higher gains. For instance, reasoning from *Qwen3-32B* produced more stable results than reasoning from *Qwen3-235B-A22B-Thinking-2507* (Table 5.3, Figure 5.6).

Additionally, reasoning format can destabilize training. Figure 5.7 shows that LoRA SFT on the v1 repair dataset with reasoning from *GPT-5-mini* resulted in lower checkpoint scores compared to reasoning

from *Qwen3-8B*. We attribute this to the structured Markdown Chain-of-Thought format used by *GPT-5-mini* (see Listing C.5), which lacks the iterative backtracking and reflections typical of the *Qwen3* family.



(a) LoRA SFT on repair dataset (v1 dataset) augmented with distilled reasoning from *Qwen3-8B* with priming.

(b) LoRA SFT on repair dataset (v4 dataset) augmented with distilled reasoning from *GPT-5-mini* and evaluated in the thinking regime.

Figure 5.7: Comparison of LoRA SFT with distilled reasoning. Left: v1 repair dataset using primed reasoning from *Qwen3-8B*. Right: v4 repair dataset with distilled reasoning from *GPT-5-mini* evaluated in *thinking* mode.

5.3.1. Reasoning Quality and Teacher Alignment

To quantify how reasoning SFT changes model behavior, we compute similarity metrics between student and teacher reasoning, as well as between generated and reference code. Table 5.4 reports *ROUGE-L*, *4-gram Jaccard* similarity, and *CodeBLEU* for several configurations, comparing student outputs against both *Qwen3-8B* (baseline) and *Qwen3-32B* (teacher).

| Model | vs Qwen3-8B | | | vs Qwen3-32B | | |
|---------------------------------|-------------|---------|----------|--------------|---------|----------|
| | ROUGE-L | Jaccard | CodeBLEU | ROUGE-L | Jaccard | CodeBLEU |
| Base Qwen3-8B | — | — | — | 0.33 | 0.07 | 0.60 |
| LoRA SFT on v4 non-reasoning | 0.35 | 0.08 | 0.60 | 0.31 | 0.06 | 0.56 |
| Self-reasoning on v4 | 0.34 | 0.09 | 0.58 | 0.29 | 0.05 | 0.55 |
| Reasoning from Qwen3-32B on v4 | 0.32 | 0.07 | 0.60 | 0.31 | 0.06 | 0.58 |
| Hybrdi reasoning from Qwen3-32B | 0.32 | 0.07 | 0.60 | 0.32 | 0.06 | 0.58 |

Table 5.4: Alignment of student models with teacher reasoning, measured using *ROUGE-L*, *Jaccard*, and *CodeBLEU*. Scores are reported relative to the base model (*Qwen3-8B*) and teacher model (*Qwen3-32B*). Higher values indicate greater similarity to the teacher. The *Qwen3-8B* base model serves as the foundation for all fine-tuned variants.

| Model | Inference mode | CodeBLEU \uparrow | EM [%] \uparrow |
|-------------------------------------|----------------|---------------------|-------------------|
| Qwen3-8B (base) | think | 0.55 | 34.74 |
| Qwen3-32B (teacher) | think | 0.52 | 38.70 |
| Qwen3-8B-v4-non_reasoning (LoRA) | think | 0.57 | 44.92 |
| Qwen3-8B-v4-non_reasoning (LoRA) | no-think | 0.54 | 44.52 |
| Qwen3-8B-v4-32B-reasoning (LoRA) | no-think | 0.51 | 35.44 |
| Qwen3-8B-v4-32B-reasoning (LoRA) | think | 0.53 | 32.21 |
| Qwen3-8B-v4-32B-reasoning (full FT) | think | 0.52 | 29.20 |

Table 5.5: *CodeBLEU* and *ExactMatch* of model-generated methods compared to ground-truth Django implementations. Model names follow the format <base model>-<dataset>-<reasoning approach> (fine-tuning method). Higher values indicate better alignment with the reference implementations.

Across configurations, scores exhibit only minor variation. Both *ROUGE – L* and *Jaccard* remain low and nearly constant, indicating limited surface-level overlap between student and teacher reasoning. *CodeBLEU* shows similarly marginal changes. These results suggest that reasoning SFT does not induce the student to replicate the teacher’s reasoning verbatim; rather, the student largely retains its own reasoning style, with only modest influence from the teacher. Additionally, the teacher’s reasoning already closely resembles the student’s natural approach.

Code-level metrics against the ground-truth repository methods reveal a distinct trend. As shown in Table 5.5, the non-reasoning APR SFT model (*Qwen3-8B-v4-nothink*) achieves the highest *CodeBLEU* and *ExactMatch* scores, surpassing both the base 8B model and the 32B teacher. In contrast, models trained on reasoning trajectories—whether purely or in hybrid form—consistently exhibit lower *CodeBLEU* and *EM*, with the largest drops observed for full fine-tuning on reasoning traces. These results indicate that incorporating reasoning during SFT does not necessarily improve alignment with ground-truth methods and can even reduce exact reproduction of the target code.

Overall, training on concise repair trajectories without explicit reasoning is the most effective way to align the student with the codebase. Reasoning SFT mainly changes the style of reasoning and code and tends to increase lexical distance from the ground truth, without yielding higher *Pass@5* or stronger code-level similarity to the reference implementations.

5.3.2. Token Efficiency Gains

A key motivation for reasoning distillation was to shorten reasoning traces while preserving repair quality. Table 5.6 reports the average and maximum reasoning lengths for several configurations.

The base *Qwen3-8B* model produces long reasoning, with 1662 words and 10931 tokens on average. Training only on non-reasoning repair trajectories already compresses reasoning substantially when the model is prompted to think: *v4 no-think* SFT reduces the average reasoning length to 1147 words and 7739 tokens, roughly a 30% reduction in tokens.

Reasoning SFT further decreases reasoning length. Training on 32B teacher reasoning reduces the average token count by about 46%, while hybrid reasoning achieves a 50% reduction with. Short-reasoning SFT produces the largest reduction, about 85%, while still maintaining performance (73.3% *Pass@5* in *no-think* mode, Table 5.3).

This reduction in reasoning length is also visible in model outputs. Listing C.1 shows the reasoning trace of the base *Qwen3-8B*, characterized by forming guesses, checking them against evidence, and revising, but it is lengthy and prone to overthinking. Our teacher model, *Qwen3-32B*, exhibits a similar reasoning process but tends to be shorter (Listing C.2). When the base model is trained with distilled reasoning from *Qwen3-32B* (Listing C.3), reasoning becomes shorter with less overthinking. Similarly, the model trained on short-reasoning traces (Listing C.4) maintains concise reasoning while preserving repair behavior.

Taken together, the reasoning experiments indicate that: (i) explicit reasoning supervision does not improve *Pass@5* beyond strong non-reasoning SFT, (ii) it can substantially reduce the length of reasoning traces, and (iii) most of the benchmark improvement arises from teaching the model to repair

| Model Configuration | Avg Words | Avg Tokens | Max Words | Count |
|----------------------------|-------------|-------------|-----------|-------|
| Base Qwen3-8B | 1662 | 10931 | 5768 | 861 |
| v4 nothink SFT | 1147 (-31%) | 7739 (-29%) | 5847 | 792 |
| 32B v4 reasoning | 901 (-46%) | 5924 (-46%) | 6267 | 804 |
| 32B v4 reasoning + nothink | 827 (-50%) | 5446 (-50%) | 4898 | 797 |
| 32B v4 short reasoning | 242 (-85%) | 1640 (-85%) | 1136 | 894 |

Table 5.6: Reasoning token statistics across training configurations (*think*-mode inference on the benchmark). Percentage reductions relative to the base model are shown in parentheses.

code using buggy methods and tests, rather than from instructing it to verbalize its reasoning.

5.4. Error Analysis

To assess the impact of our training data, it is important to examine which types of errors the model is able to resolve and which remain challenging. We analyse unresolved failures by grouping cases that persist beyond turn 5 according to their exception type. Figure 5.8 reports both the number of unresolved errors and the total errors caused across four configurations: base student *Qwen3-8B*, base teacher *Qwen3-32B*, *Qwen3-8B* trained on *v4* non-reasoning dataset, and *Qwen3-8B* SFTed on *v4* reasoning dataset.

The fine-tuned model trained on non-reasoning repair trajectories improves repair efficiency over the base *Qwen3-8B* model, generating fewer intermediate errors per solved instance. However, the total number of unresolved cases remains largely unchanged across all configurations. *AssertionError* is the dominant remaining error type, indicating that most failures occur from incorrect functional behavior: the methods run but produce wrong outputs. This highlights that addressing the hardest cases requires deeper domain knowledge of the codebase, which isolated method-level repair trajectories do not fully provide.

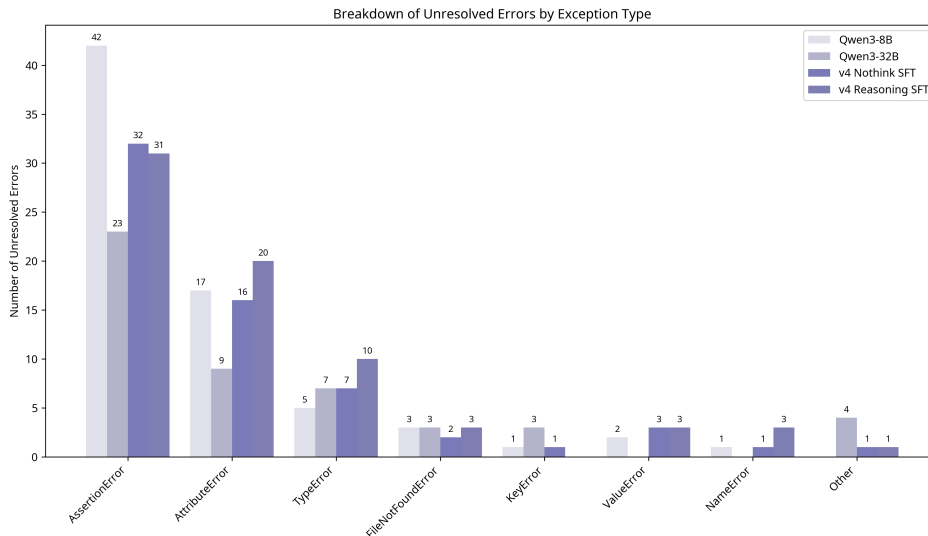


Figure 5.8: Breakdown of unresolved errors (after turn 5) by exception type: base student *Qwen3-8B*, base teacher *Qwen3-32B*, *Qwen3-8B* trained on *v4* non-reasoning dataset, and *Qwen3-8B* SFTed on *v4* reasoning dataset.

5.5. Generalization Across Model Families

To evaluate generalization of our experiments across model families and sizes, we fine-tune *Qwen3-14B*, *Qwen3-32B*, *Llama-3.1-8B-Instruct*, and *Mistral-8B-Instruct-2410* on *v4* using LoRA ($r = 32$) (results are in Table 5.7).

All models benefit from APR fine-tuning, with the largest gains in smaller models: *Llama* improves from 63.0% to 72.5% (+9.5 points) and *Mistral* from 59.7% to 74.3% (+14.6 points) in think mode. Within the *Qwen3* family, the 14B gains +3.5, and 32B gains +2.0.

Figures B.3 and B.4 show performance across epochs for *Mistral-8B-Instruct-2410* and *Llama-3.1-8B-Instruct*, revealing significant improvements in both method generation and repair capability. LoRA training yields larger gains than FFT, suggesting potential overfitting in the latter. While format learning may contribute to these improvements, we observe no format deviations from base models during evaluation.

Fine-tuned *Qwen3-8B* matches base *Qwen3-14B* performance, and fine-tuned *Qwen3-14B* reaches base *Qwen3-32B* levels. Though *Qwen3-14B* shows continued repair improvement in later turns (Figure B.4), the diminishing returns in larger models suggest benchmark saturation.

| Model | Think Mode | | | Nothink Mode | | |
|--------------------------|------------|-------|--------|--------------|-------|-------|
| | Base | SFT | Gain | Base | SFT | Gain |
| Qwen3-8B | 76.5% | 82.4% | +5.9% | 69.9% | 78.1% | +8.2% |
| Qwen3-14B | 81.4% | 84.9% | +3.5% | 75.8% | 82.1% | +6.3% |
| Qwen3-32B | 84.5% | 86.6% | +2.0% | 77.7% | 80.9% | +3.2% |
| Llama-3.1-8B-Instruct | 63.0% | 72.5% | +9.5% | — | — | — |
| Mistral-8B-Instruct-2410 | 59.7% | 74.3% | +14.6% | — | — | — |

Table 5.7: APR distillation generalizes across model families with gains inversely proportional to model size (smaller models benefit most). All models are trained with LoRA ($r = 32$) on v4 dataset.

5.5.1. Out-of-Domain Performance: SWE-bench

To assess whether repository-specific APR fine-tuning degrades general repair capabilities, we evaluate selected *Qwen3-8B* checkpoints on SWE-Agent, where the model fixes GitHub issues using available tools (e.g., string replace, search, inspect). The benchmark contains 220 Django repository issues, enabling evaluation in a different setup within the same domain. Results in Table 5.8 show that LoRA-based checkpoint, trained on repair examples, maintains resolved rates near the base model, however training on both repair and method generation trajectories decreases the score. This observation hints that the choice of the fine-tuning method is not the single factor that prevents out-of-domain performance, but also the characteristics of the training dataset. In this example, training on two tasks, complementary in essence, seems to contribute to forgetting the previous agentic or tool-calling behavior of the model.

Furthermore, FFT reduces the resolved rate to 9.6%. This gap supports prior findings that parameter-efficient adaptation mitigates forgetting compared to full model updates [7].

| Configuration | Resolved |
|---|----------|
| Qwen3-8B base | 20% |
| Qwen3-8B SFTed on v4 non-reasoning LoRA (epoch 8) | 17% |
| Qwen3-8B SFTed on v5 non-reasoning LoRA (epoch 7) | 13% |
| Qwen3-8B SFTed on v5 non-reasoning FFT (epoch 9) | 9.6% |

Table 5.8: Out-of-domain repair on 220 Django issues from SWE-bench. LoRA preserves most of the base model’s performance, while FFT induces substantial catastrophic forgetting in the agentic setting.

6

Discussion

This chapter synthesizes the empirical findings presented in Chapter 5 and discusses their implications for domain-specific fine-tuning of SLMs for APR. We revisit the research questions in light of the observed results, outline practical recommendations for practitioners, and reflect on the limitations of the proposed approach. Finally, we identify promising directions for future work, particularly beyond fine-tuning alone.

6.1. Answer to research questions

In this section, we directly address the research questions introduced earlier. We summarize the key results and distill practical insights on dataset construction, fine-tuning strategies, and reasoning supervision for in-domain APR.

6.1.1. Factors for Efficient and Cost-Effective In-Domain APR Fine-Tuning

Our results show that effective in-domain APR fine-tuning is primarily driven by dataset scale and task diversity. Performance improves steadily as the dataset evolves, where additional teacher models, ground-truth solutions, and failed trajectories are incorporated. Adding complementary tasks, such as single-turn method generation, further improves repair accuracy and stabilizes performance in the *no-think* regime. Overall, fine-tuning yields clear gains over the base model. *Qwen3-8B* trained with LoRA improves by up to 5.9% in *think* mode and 8.2% in *no-think* mode, while FFT reaches gains of 4.3% and 9.7% respectively.

Comparing fine-tuning strategies highlights a trade-off between efficiency and scale. LoRA achieves the highest *think*-mode performance, reaching 82.4% $Pass@5$, while requiring substantially fewer resources. In our setup, LoRA training is approximately four times faster than FFT. FFT becomes competitive only with larger and more diverse datasets, where it closes the gap between *think* and *no-think* regimes, achieving 79.6% in both modes. Hyperparameter analysis shows that LoRA ranks between 32 and 64 provide sufficient capacity for learning repository-specific patterns. Higher ranks introduce instability without performance gains, and training typically converges after 8 epochs.

Analysis across repair turns shows that most successful repairs occur within the first two turns, with limited gains afterward. LoRA yields more consistent improvements across turns, while FFT shows higher variance before stabilizing. Importantly, both strategies reduce the performance gap between *think* and *no-think* inference, indicating effective transfer of repair knowledge. Beyond this, we observe that LoRA consistently outperforms FFT across other model families and sizes, as well as in agentic tool-calling settings, demonstrating robustness to overfitting and strong generalization. Taken together, these results suggest that LoRA with moderate rank, combined with diverse repair and generation data, provides the most cost-effective, stable, and broadly applicable fine-tuning pipeline for in-domain APR on SLMs.

6.1.2. Effects of Reasoning Supervision on Repair Performance

Our experiments show that explicit reasoning supervision does not improve repair performance beyond strong non-reasoning APR SFT. Models trained only on reasoning trajectories consistently underperform non-reasoning counterparts, and none exceed the 82.4% *Pass@5* achieved by the non-reasoning model in *think* mode. Hybrid training that mixes reasoning and non-reasoning examples recovers most of the lost performance but does not surpass the best non-reasoning configurations. The strongest hybrid model reaches 79.8% in *think* mode and 79.6% in *no-think* mode, comparable to full fine-tuning without reasoning. These results indicate that the primary learning signal for APR comes from exposure to buggy methods, tests, and correct implementations rather than from explicit reasoning traces.

The effectiveness of reasoning supervision depends strongly on teacher alignment. Reasoning distilled from teacher models closer in family and size to the student leads to more stable training, while reasoning from larger or different teachers often degrades performance. This sensitivity appears to stem from two factors. First, long reasoning traces dominate the training signal and can reduce focus on learning the corrected code. Second, differences in reasoning structure and format can destabilize learning even when the reasoning content is correct. Alignment metrics confirm that students largely retain their original reasoning style, and models trained without reasoning supervision achieve the highest *CodeBLEU* and exact match scores against ground-truth methods.

While reasoning supervision does not improve repair accuracy, it provides clear benefits for inference efficiency. Non-reasoning SFT already shortens reasoning traces by about 30% relative to the base model. Reasoning distillation further reduces reasoning length by 46% using standard teacher reasoning and by up to 85% with short-reasoning supervision, while maintaining reasonable repair performance. This compression mitigates overthinking in SLMs and substantially lowers inference cost.

Overall, the most effective strategy for in-domain APR is to prioritize concise, non-reasoning repair trajectories that directly teach code correction. Reasoning supervision is best used selectively, primarily for reducing reasoning length when teacher and student models are well aligned, rather than as a mechanism for improving repair accuracy.

6.2. Limitations and Threats to Validity

This study has several limitations that constrain the scope and generality of our findings. First, all experiments are conducted on a single repository, Django. While we observe consistent trends across model families and sizes, these results may not directly transfer to other repositories, programming languages, or domains with different coding styles, test structures, or failure modes.

Second, we focus exclusively on method-level APR. This scope enables the model to learn a wide range of practical bugs within individual methods, but it does not cover repairs that require broader context, such as cross-method dependencies, architectural constraints, or multi-file changes. The effectiveness of our dataset construction strategy and fine-tuning choices may therefore differ for tasks that require longer-range context.

Third, our conclusions are conditioned on the specific model families and sizes evaluated in this work. While we included multiple open-source architectures and sizes, we did not explore larger models across different families, and the observed trade-offs between fine-tuning strategies, dataset composition, and reasoning supervision may differ in those settings.

Fourth, the design of prompts used for reasoning elicitation introduces an additional source of variance. Reasoning structure, formatting, and verbosity are influenced by prompt phrasing, and different elicitation strategies may lead to different training dynamics or outcomes. While we control prompts across experiments, our results may not generalize to alternative reasoning formulations or prompting styles.

Finally, our exploration of the training space is necessarily incomplete. Although we systematically study dataset size, fine-tuning method, and LoRA rank, other factors such as learning rates, adapter placement, optimizer choices, and alternative parameter-efficient methods may further affect performance and stability.

Beyond these experimental constraints, our results highlight a broader limitation of supervised fine-tuning as a standalone adaptation mechanism. While fine-tuning effectively transfers repository-specific

patterns when high-quality data is available, it remains sensitive to data coverage and diversity. Hard repair cases that require deeper semantic or system-level understanding remain unresolved, suggesting that fine-tuning alone is insufficient to address the most challenging bugs.

6.3. Future Work: Context-Aware Training and Complementary Tasks

Our findings indicate that robust, domain-specific APR systems require more than fine-tuning alone. A promising direction is to combine fine-tuned models with agentic systems that leverage structured, task-aware contexts [25]. Context engineering provides access to relevant code, tests, documentation, execution feedback, and external tools, enabling models to reason about functional behavior and system-level constraints despite the limited parametric capacity of SLMs. Fine-tuning supplies domain familiarity, while context engineering delivers the missing state and guidance, together forming a more flexible and scalable solution.

Beyond context, structured training across complementary code tasks can improve generalization and reduce task interference. Strong performance on a single repository or benchmark does not guarantee robustness in agentic or tool-assisted scenarios. Future work should explore:

- **Task structure and ordering.** Investigate how different code tasks interact during training and how sequencing affects knowledge transfer and forgetting.
- **Curriculum-based agent training.** Gradually expose models to tasks of increasing contextual scope, from method-level repair to class-level reasoning and multi-step, tool-assisted debugging.

Combining context-aware inference with structured, complementary task training is a promising solution to make SLMs more adaptable and reliable in real-world deployments.

7

Conclusion

This work studied how SLMs can be adapted for APR in repository-specific settings. We introduce a multi-teacher distillation pipeline that generates repair trajectories in a multi-turn setup, producing diverse training items that include both successful fixes and intermediate failures. Using this pipeline, we systematically analyze the impact of dataset size and diversity on fine-tuning performance, allowing us to understand how repair variety influences performance. We then evaluate supervised fine-tuning across different strategies, hyperparameters, and inference regimes to characterize performance, stability, and computational trade-offs.

Our results show that parameter-efficient fine-tuning provides a practical and effective approach for adapting SLMs to in-domain APR. In particular, LoRA delivers strong and stable performance gains at low computational cost when adapter capacity is chosen appropriately. Across datasets and tasks, LoRA ranks between 32 and 64 consistently converge well, while higher ranks provide no benefit and often introduce instability. These findings highlight the importance of careful hyperparameter selection when fine-tuning smaller models on moderate-sized, repository-specific datasets.

We further find that explicit reasoning supervision is not necessary to achieve strong repair performance. Training on concise, non-reasoning repair trajectories yields the highest accuracy and the strongest alignment with ground-truth code. While reasoning distillation does not improve benchmark performance, it plays an important role in efficiency: distilled reasoning substantially shortens reasoning traces, by up to 85%, without harming repair accuracy. This makes reasoning supervision useful as a compression tool when reasoning output is required but inference cost must be controlled.

Our experiments also show that the proposed approach generalizes across model families, with the largest gains observed in smaller models. LoRA-based fine-tuning preserves general repair capability for in-domain agentic settings, while full fine-tuning leads to significant forgetting. Error analysis indicates that the remaining failures are dominated by functional correctness errors, suggesting that method-level repair trajectories alone are insufficient for the hardest cases.

Overall, this study provides practical evidence that SLMs can be effectively deployed for real-world program repair when supported by careful data construction and lightweight fine-tuning. Although SLMs lack the capacity of larger models, their advantages in cost, latency, and deployability make them well suited for practical systems. Future work should explore integrating fine-tuned SLMs into agentic frameworks with richer context, tool use, and complementary training tasks, which may be necessary to address the remaining hard repair cases and move beyond current benchmark limits.

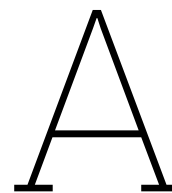
References

- [1] Rishabh Agarwal et al. *On-Policy Distillation of Language Models: Learning from Self-Generated Mistakes*. 2024. arXiv: 2306.13649 [cs.LG]. URL: <https://arxiv.org/abs/2306.13649>.
- [2] Wasi Uddin Ahmad et al. *OpenCodeInstruct: A Large-scale Instruction Tuning Dataset for Code LLMs*. 2025. arXiv: 2504.04030 [cs.SE]. URL: <https://arxiv.org/abs/2504.04030>.
- [3] Anthropic. *Building effective agents*. Anthropic Engineering Blog. <https://www.anthropic.com/engineering/building-effective-agents>. Dec. 2024.
- [4] Anthropic. *The Claude 3 Model Family: Opus, Sonnet, Haiku*. Anthropic model card. https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf. Mar. 2024.
- [5] Jacob Austin et al. *Program Synthesis with Large Language Models*. 2021. arXiv: 2108.07732 [cs.PL]. URL: <https://arxiv.org/abs/2108.07732>.
- [6] Peter Belcak et al. *Small Language Models are the Future of Agentic AI*. 2025. arXiv: 2506.02153 [cs.AI]. URL: <https://arxiv.org/abs/2506.02153>.
- [7] Dan Biderman et al. *LoRA Learns Less and Forgets Less*. 2024. arXiv: 2405.09673 [cs.LG]. URL: <https://arxiv.org/abs/2405.09673>.
- [8] Federico Cassano et al. *MultiPL-E: A Scalable and Extensible Approach to Benchmarking Neural Code Generation*. 2022. arXiv: 2208.08227 [cs.LG]. URL: <https://arxiv.org/abs/2208.08227>.
- [9] Aolin Chen et al. *Studying and Understanding the Effectiveness and Failures of Conversational LLM-Based Repair*. 2025. arXiv: 2503.15050 [cs.SE]. URL: <https://arxiv.org/abs/2503.15050>.
- [10] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: 2107.03374 [cs.LG]. URL: <https://arxiv.org/abs/2107.03374>.
- [11] Jonathan Cordeiro, Shayan Noei, and Ying Zou. "LLM-Driven Code Refactoring: Opportunities and Limitations". In: *2025 IEEE/ACM Second IDE Workshop (IDE)*. 2025, pp. 32–36. DOI: 10.1109/IDE66625.2025.00011.
- [12] DeepSeek-AI et al. *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*. 2025. arXiv: 2501.12948 [cs.CL]. URL: <https://arxiv.org/abs/2501.12948>.
- [13] Tim Dettmers et al. *QLoRA: Efficient Finetuning of Quantized LLMs*. 2023. arXiv: 2305.14314 [cs.LG]. URL: <https://arxiv.org/abs/2305.14314>.
- [14] Tuan Dinh et al. *Large Language Models of Code Fail at Completing Code with Potential Bugs*. 2023. arXiv: 2306.03438 [cs.LG]. URL: <https://arxiv.org/abs/2306.03438>.
- [15] Kanishk Gandhi et al. *Cognitive Behaviors that Enable Self-Improving Reasoners, or, Four Habits of Highly Effective STaRs*. 2025. arXiv: 2503.01307 [cs.CL]. URL: <https://arxiv.org/abs/2503.01307>.
- [16] Jonas Gehring et al. "RLEF: Grounding Code LLMs in Execution Feedback with Reinforcement Learning". In: *Proceedings of the 42nd International Conference on Machine Learning*. Vol. 267. Proceedings of Machine Learning Research. 2025, pp. 19034–19055. URL: <https://proceedings.mlr.press/v267/gehring25a.html>.
- [17] Aaron Grattafiori et al. *The Llama 3 Herd of Models*. 2024. arXiv: 2407.21783 [cs.AI]. URL: <https://arxiv.org/abs/2407.21783>.
- [18] Alex Gu et al. "CRUXEval: a benchmark for code reasoning, understanding and execution". In: *Proceedings of the 41st International Conference on Machine Learning*. ICML'24. Vienna, Austria: JMLR.org, 2024.

- [19] Xiaodong Gu et al. “On the Effectiveness of Large Language Models in Domain-Specific Code Generation”. In: *ACM Trans. Softw. Eng. Methodol.* 34.3 (Feb. 2025). ISSN: 1049-331X. DOI: [10.1145/3697012](https://doi-org.tudelft.idm.oclc.org/10.1145/3697012). URL: <https://doi-org.tudelft.idm.oclc.org/10.1145/3697012>.
- [20] Xinyi Hou et al. “Large Language Models for Software Engineering: A Systematic Literature Review”. In: *ACM Trans. Softw. Eng. Methodol.* 33.8 (Dec. 2024). ISSN: 1049-331X. DOI: [10.1145/3695988](https://doi.org/10.1145/3695988). URL: <https://doi.org/10.1145/3695988>.
- [21] Edward J. Hu et al. *LoRA: Low-Rank Adaptation of Large Language Models*. 2021. arXiv: [2106.09685](https://arxiv.org/abs/2106.09685) [cs.CL]. URL: <https://arxiv.org/abs/2106.09685>.
- [22] Kai Huang et al. *A Survey on Automated Program Repair Techniques*. 2023. arXiv: [2303.18184](https://arxiv.org/abs/2303.18184) [cs.SE]. URL: <https://arxiv.org/abs/2303.18184>.
- [23] Mathav Raj J et al. *Fine Tuning LLM for Enterprise: Practical Guidelines and Recommendations*. 2024. arXiv: [2404.10779](https://arxiv.org/abs/2404.10779) [cs.SE]. URL: <https://arxiv.org/abs/2404.10779>.
- [24] Naman Jain et al. *LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code*. 2024. arXiv: [2403.07974](https://arxiv.org/abs/2403.07974) [cs.SE]. URL: <https://arxiv.org/abs/2403.07974>.
- [25] Yichao Ji. *Context Engineering for AI Agents: Lessons from Building Manus*. Manus Blog. July 2025. URL: <https://manus.im/blog/Context-Engineering-for-AI-Agents-Lessons-from-Building-Manus>.
- [26] Albert Q. Jiang et al. *Mistral 7B*. 2023. arXiv: [2310.06825](https://arxiv.org/abs/2310.06825) [cs.CL]. URL: <https://arxiv.org/abs/2310.06825>.
- [27] Juyong Jiang et al. “A Survey on Large Language Models for Code Generation”. In: *ACM Trans. Softw. Eng. Methodol.* (July 2025). Just Accepted. ISSN: 1049-331X. DOI: [10.1145/3747588](https://doi-org.tudelft.idm.oclc.org/10.1145/3747588). URL: <https://doi-org.tudelft.idm.oclc.org/10.1145/3747588>.
- [28] Carlos E. Jimenez et al. *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?* 2024. arXiv: [2310.06770](https://arxiv.org/abs/2310.06770) [cs.CL]. URL: <https://arxiv.org/abs/2310.06770>.
- [29] René Just, Darioush Jalali, and Michael D. Ernst. “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*. ISSSTA 2014. San Jose, CA, USA: Association for Computing Machinery, 2014, pp. 437–440. ISBN: 9781450326452. DOI: [10.1145/2610384.2628055](https://doi-org.tudelft.idm.oclc.org/10.1145/2610384.2628055). URL: <https://doi-org.tudelft.idm.oclc.org/10.1145/2610384.2628055>.
- [30] Sungmin Kang et al. *Evaluating Diverse Large Language Models for Automatic and General Bug Reproduction*. 2023. arXiv: [2311.04532](https://arxiv.org/abs/2311.04532) [cs.SE]. URL: <https://arxiv.org/abs/2311.04532>.
- [31] Roham Koohestani, Philippe de Bekker, and Maliheh Izadi. *Benchmarking AI Models in Software Engineering: A Review, Search Tool, and Enhancement Protocol*. 2025. arXiv: [2503.05860](https://arxiv.org/abs/2503.05860) [cs.SE]. URL: <https://arxiv.org/abs/2503.05860>.
- [32] Jia Li et al. “EvoCodeBench: an evolving code generation benchmark with domain-specific evaluations”. In: *Proceedings of the 38th International Conference on Neural Information Processing Systems*. NIPS ’24. Vancouver, BC, Canada: Curran Associates Inc., 2025. ISBN: 9798331314385.
- [33] Raymond Li et al. *StarCoder: may the source be with you!* 2023. arXiv: [2305.06161](https://arxiv.org/abs/2305.06161) [cs.CL]. URL: <https://arxiv.org/abs/2305.06161>.
- [34] Jiatae Liu et al. *RLTF: Reinforcement Learning from Unit Test Feedback*. 2023. arXiv: [2307.04349](https://arxiv.org/abs/2307.04349) [cs.AI]. URL: <https://arxiv.org/abs/2307.04349>.
- [35] Jiawei Liu et al. *Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation*. 2023. arXiv: [2305.01210](https://arxiv.org/abs/2305.01210) [cs.SE]. URL: <https://arxiv.org/abs/2305.01210>.
- [36] Ilya Loshchilov and Frank Hutter. *Decoupled Weight Decay Regularization*. 2019. arXiv: [1711.05101](https://arxiv.org/abs/1711.05101) [cs.LG]. URL: <https://arxiv.org/abs/1711.05101>.
- [37] Aman Madaan et al. *Self-Refine: Iterative Refinement with Self-Feedback*. 2023. arXiv: [2303.17651](https://arxiv.org/abs/2303.17651) [cs.CL]. URL: <https://arxiv.org/abs/2303.17651>.

- [38] Jiya Manchanda et al. *The Open Source Advantage in Large Language Models (LLMs)*. 2025. arXiv: 2412.12004 [cs.CL]. URL: <https://arxiv.org/abs/2412.12004>.
- [39] Mohammad Mahdi Mohajer et al. *SkipAnalyzer: A Tool for Static Code Analysis with Large Language Models*. 2023. arXiv: 2310.18532 [cs.SE]. URL: <https://arxiv.org/abs/2310.18532>.
- [40] Erik Nijkamp et al. *CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis*. 2023. arXiv: 2203.13474 [cs.LG]. URL: <https://arxiv.org/abs/2203.13474>.
- [41] Theo X. Olausson et al. *Is Self-Repair a Silver Bullet for Code Generation?* 2024. arXiv: 2306.09896 [cs.CL]. URL: <https://arxiv.org/abs/2306.09896>.
- [42] OpenAI et al. *GPT-4 Technical Report*. 2024. arXiv: 2303.08774 [cs.CL]. URL: <https://arxiv.org/abs/2303.08774>.
- [43] Qwen Team. *Qwen3-Coder: Agentic Coding in the World*. July 2025. URL: <https://qwen.ai/blog?id=d927d7d2e59d059045ce758ded34f98c0186d2d7&from=research.research-list>.
- [44] Baptiste Rozière et al. *Code Llama: Open Foundation Models for Code*. 2024. arXiv: 2308.12950 [cs.CL]. URL: <https://arxiv.org/abs/2308.12950>.
- [45] Anton Semenkin and Michelle Frost. *Mellum Goes Open Source: A Purpose-Built LLM for Developers, Now on Hugging Face*. JetBrains AI Blog. <https://blog.jetbrains.com/ai/2025/04/mellum-goes-open-source-a-purpose-built-llm-for-developers-now-on-hugging-face/>. Apr. 2025.
- [46] Jiho Shin et al. *Domain Adaptation for Code Model-based Unit Test Case Generation*. 2024. arXiv: 2308.08033 [cs.SE]. URL: <https://arxiv.org/abs/2308.08033>.
- [47] Noah Shinn et al. *Reflexion: Language Agents with Verbal Reinforcement Learning*. 2023. arXiv: 2303.11366 [cs.AI]. URL: <https://arxiv.org/abs/2303.11366>.
- [48] Dominik Sobania et al. *An Analysis of the Automatic Bug Fixing Performance of ChatGPT*. 2023. arXiv: 2301.08653 [cs.SE]. URL: <https://arxiv.org/abs/2301.08653>.
- [49] Weisong Sun et al. *Source Code Summarization in the Era of Large Language Models*. 2024. arXiv: 2407.07959 [cs.SE]. URL: <https://arxiv.org/abs/2407.07959>.
- [50] Xunzhu Tang, Jacques Klein, and Tegawendé F Bissyandé. “Boosting Open-Source LLMs for Program Repair via Reasoning Transfer and LLM-Guided Reinforcement Learning”. In: *arXiv preprint arXiv:2506.03921* (2025).
- [51] Gemini Team et al. *Gemini: A Family of Highly Capable Multimodal Models*. 2025. arXiv: 2312.11805 [cs.CL]. URL: <https://arxiv.org/abs/2312.11805>.
- [52] Yun-Da Tsai, Mingjie Liu, and Haoxing Ren. *Code Less, Align More: Efficient LLM Fine-tuning for Code Generation with Data Pruning*. 2024. arXiv: 2407.05040 [cs.SE]. URL: <https://arxiv.org/abs/2407.05040>.
- [53] Zengzhi Wang et al. *OctoThinker: Mid-training Incentivizes Reinforcement Learning Scaling*. 2025. arXiv: 2506.20512 [cs.CL]. URL: <https://arxiv.org/abs/2506.20512>.
- [54] Jason Wei et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. arXiv: 2201.11903 [cs.CL]. URL: <https://arxiv.org/abs/2201.11903>.
- [55] Kyle Wong et al. *Investigating the Transferability of Code Repair for Low-Resource Programming Languages*. 2024. arXiv: 2406.14867 [cs.LG]. URL: <https://arxiv.org/abs/2406.14867>.
- [56] Xiaohan Xu et al. *A Survey on Knowledge Distillation of Large Language Models*. 2024. arXiv: 2402.13116 [cs.CL]. URL: <https://arxiv.org/abs/2402.13116>.
- [57] An Yang et al. *Qwen3 Technical Report*. 2025. arXiv: 2505.09388 [cs.CL]. URL: <https://arxiv.org/abs/2505.09388>.
- [58] Boyang Yang et al. “MOREpair: Teaching LLMs to Repair Code via Multi-Objective Fine-Tuning”. In: *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*. Use official venue details once available. 2024.
- [59] Boyang Yang et al. “A Survey of LLM-based Automated Program Repair: Taxonomies, Design Paradigms, and Applications”. In: *arXiv preprint arXiv:2506.23749* (2025). URL: <https://arxiv.org/abs/2506.23749>.

- [60] Boyang Yang et al. *MORepair: Teaching LLMs to Repair Code via Multi-Objective Fine-tuning*. 2025. arXiv: [2404.12636](https://arxiv.org/abs/2404.12636) [cs.SE]. URL: <https://arxiv.org/abs/2404.12636>.
- [61] Dayu Yang et al. *Code to Think, Think to Code: A Survey on Code-Enhanced Reasoning and Reasoning-Driven Code Intelligence in LLMs*. 2025. arXiv: [2502.19411](https://arxiv.org/abs/2502.19411) [cs.CL]. URL: <https://arxiv.org/abs/2502.19411>.
- [62] John Yang et al. *SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering*. 2024. arXiv: [2405.15793](https://arxiv.org/abs/2405.15793) [cs.SE]. URL: <https://arxiv.org/abs/2405.15793>.
- [63] Burak Yetiştirgen et al. *Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT*. 2023. arXiv: [2304.10778](https://arxiv.org/abs/2304.10778) [cs.SE]. URL: <https://arxiv.org/abs/2304.10778>.
- [64] Xin Yin et al. “ThinkRepair: Self-Directed Automated Program Repair”. In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2024. Vienna, Austria: Association for Computing Machinery, 2024, pp. 1274–1286. ISBN: 9798400706127. DOI: [10.1145/3650212.3680359](https://doi.org/10.1145/3650212.3680359). URL: <https://doi.org/10.1145/3650212.3680359>.
- [65] Bin Yu et al. *Long-Short Chain-of-Thought Mixture Supervised Fine-Tuning Eliciting Efficient Reasoning in Large Language Models*. 2025. arXiv: [2505.03469](https://arxiv.org/abs/2505.03469) [cs.CL]. URL: <https://arxiv.org/abs/2505.03469>.
- [66] Quanjun Zhang et al. “A Survey of Learning-based Automated Program Repair”. In: *ACM Transactions on Software Engineering and Methodology* 33.2 (2023), pp. 1–69. DOI: [10.1145/3631974](https://doi.org/10.1145/3631974).
- [67] Dewu Zheng et al. *Top General Performance = Top Domain Performance? DomainCodeBench: A Multi-domain Code Generation Benchmark*. 2025. arXiv: [2412.18573](https://arxiv.org/abs/2412.18573) [cs.SE]. URL: <https://arxiv.org/abs/2412.18573>.
- [68] Qinkai Zheng et al. *CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X*. 2024. arXiv: [2303.17568](https://arxiv.org/abs/2303.17568) [cs.LG]. URL: <https://arxiv.org/abs/2303.17568>.
- [69] Chunting Zhou et al. *LIMA: Less Is More for Alignment*. 2023. arXiv: [2305.11206](https://arxiv.org/abs/2305.11206) [cs.CL]. URL: <https://arxiv.org/abs/2305.11206>.
- [70] Armin Zirak and Hadi Hemmati. *Improving Automated Program Repair with Domain Adaptation*. 2022. arXiv: [2212.11414](https://arxiv.org/abs/2212.11414) [cs.SE]. URL: <https://arxiv.org/abs/2212.11414>.



Prompting Setup

A.1. Prompt Templates Used in the Django APR Benchmark

```
1 system: |
2   You are a helpful programming assistant for Python. Return code in a single code block:
3   ```python
4   CODE
5   ```
6   You can think before writing code. You can also add your thinking in the comments.
7   Do not include decorators. Do not provide usage examples.
8   Start with the method declaration.
```

Listing A.1: System prompt used across both repair trajectory distillation and evaluation specifying output format and constraints for Python method generation

```
1 base: |
2   Generate code for the method `{METHOD_NAME}` in the class `{CLASS_NAME}`.
3   Do not generate the whole class, only the target method.
4   Start with method declaration `{METHOD_DECLARATION}`.
5
6 dp_description: |
7   Description:
8
9   Class name: `{CLASS_NAME}`
10
11  Method declaration:
12  ```python
13  {METHOD_DECLARATION}```
14
15  Method description:
16  {METHOD_DESCRIPTION}
17
18  Class content:
19  ```python
20  {FILE_CONTENT}```
```

Listing A.2: Base prompt used to task the model to generate the respective method, given the class name and contents where the method resides and the method declaration including the arguments, return types and description.

```
1 iteration: |
2   Correct the answer. Tests failed:
3   TEST_RESULT:
4   {test_output}
```

Listing A.3: Prompt used during subsequent iterations of the repair task. When the model's solution fails to pass all the associated test cases, the model is tasked to repair the buggy method.

A.2. Reasoning Prompts Considered

```
1
2 Please fix the buggy code and reason in a step-by-step way.
3
4 Task description: {TASK_DESCRIPTION}
5 Buggy method: {BUGGY_CODE}
6 Test feedback: {HUMAN_FEEDBACK}
7
8 Here is the correct solution you will give: {CORRECT_SOLUTION}
9
10 <think>\n We will not think extremely long. We are given the task to fix ...
```

Listing A.4: Priming used to elicit the model's reasoning. This priming in particular showed poor results on the final SFTed model on these reasoning trajectories, which indicates that the model is fragile to different structured or formats of reasoning during training.

```
1
2 Given a task description and its correct solution, reason on how to arrive at the correct
  solution in a structured way.
3
4 When reasoning, carefully analyze why the wrong method failed, and connect this to the test
  feedback.
5
6 Highlight the specific mistakes or missing considerations, and explain how the correct
  solution addresses them.
7
8 Be explicit about what to be careful about next time, emphasizing the importance of not
  overlooking aspects of the error, double-checking reasoning, and verifying the
  correctness of the solution.
9
10 Task description: {TASK_DESCRIPTION}
11 Wrong method: {BUGGY_CODE}
12 Test feedback: {HUMAN_FEEDBACK}
13 Correct solution: {CORRECT_SOLUTION}
14
15 <think>\n We will not think too long. We are given the task to fix...
```

Listing A.5: Reflection-style reasoning prompt used to gather reasoning trajectories. The SFTed model showed poor performance on our benchmark, which shows the fragility of the model to a different format of reasoning.

B

Additional Results

B.1. Fine-tuning results of additional experiments

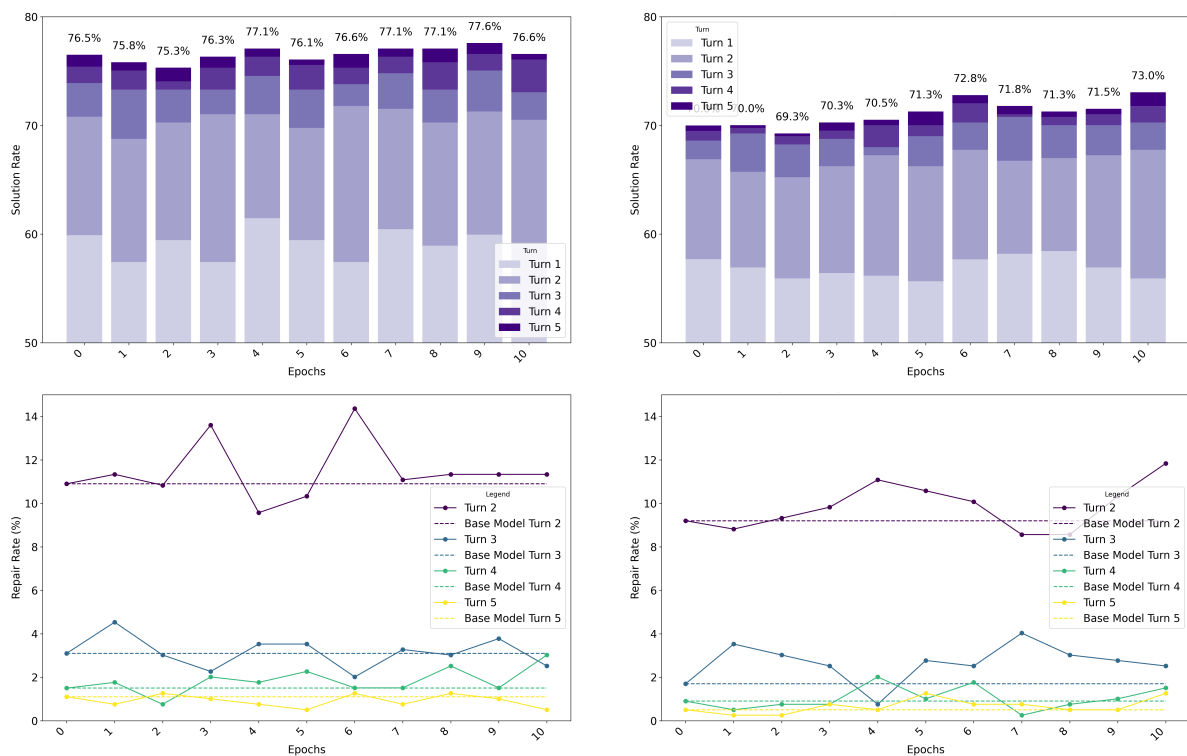
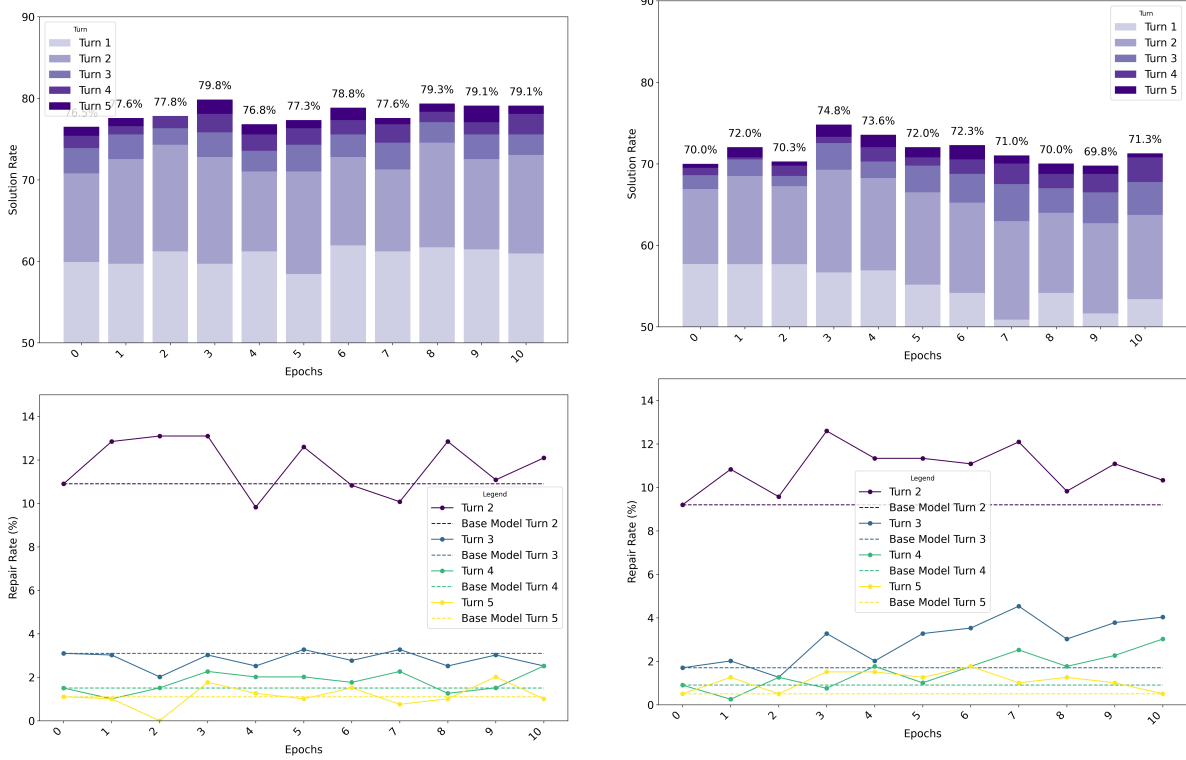
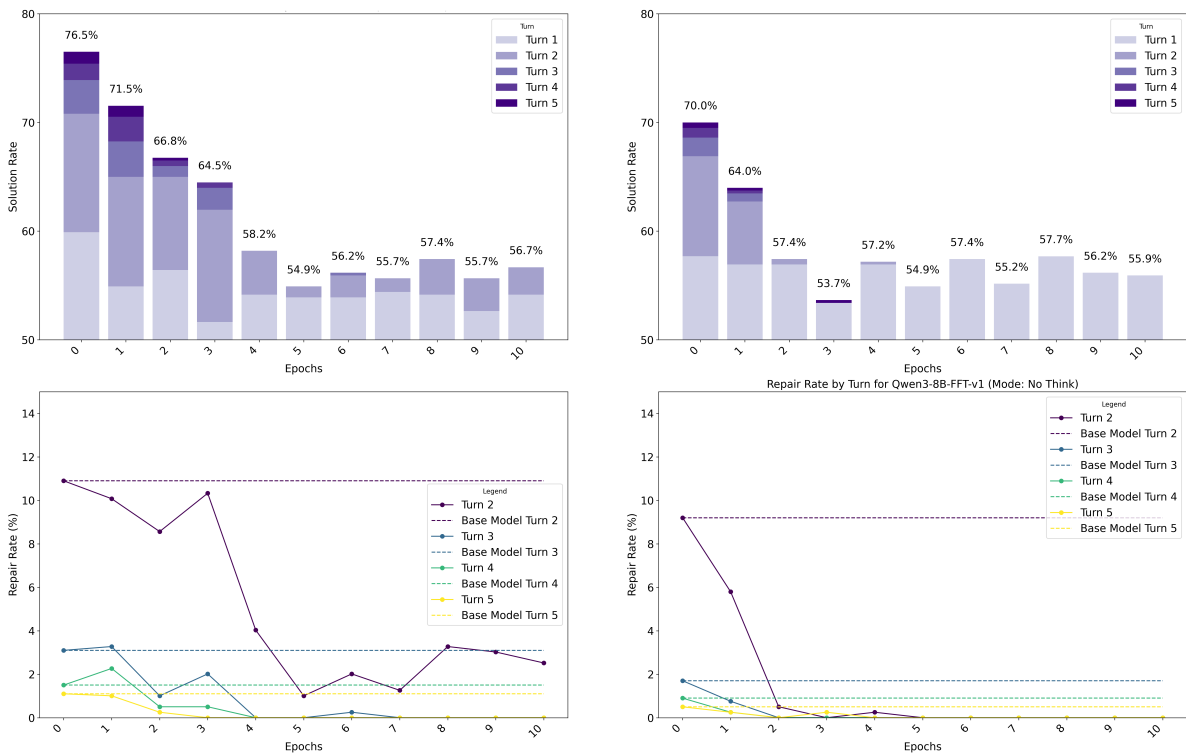


Figure B.1: Checkpoint scores and performance across turns of the *Qwen3-8B* model SFTed with LoRA on *v1* dataset across 10 epochs in two regimes: thinking (left) and no-think (right).

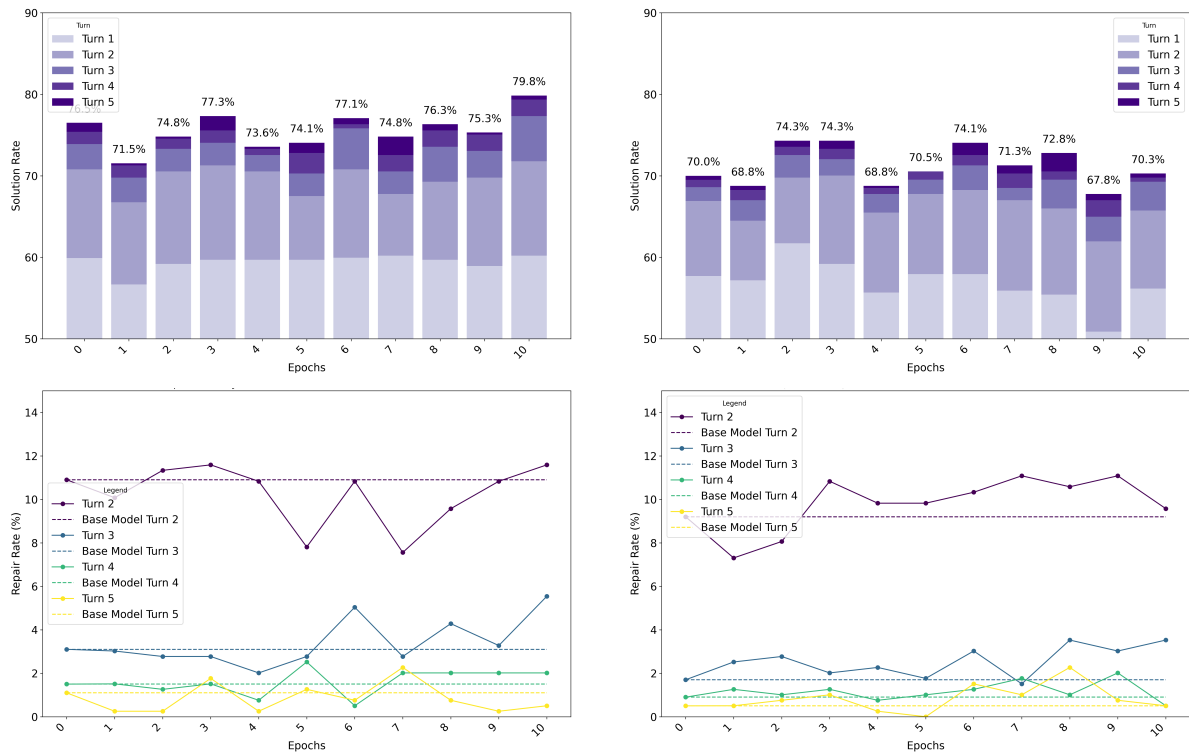


(a) Checkpoint scores and performance across turns of the *Qwen3-8B* model SFTed with **LoRA** on *v3* dataset across 10 epochs in two regimes: thinking (left) and no-think (right).

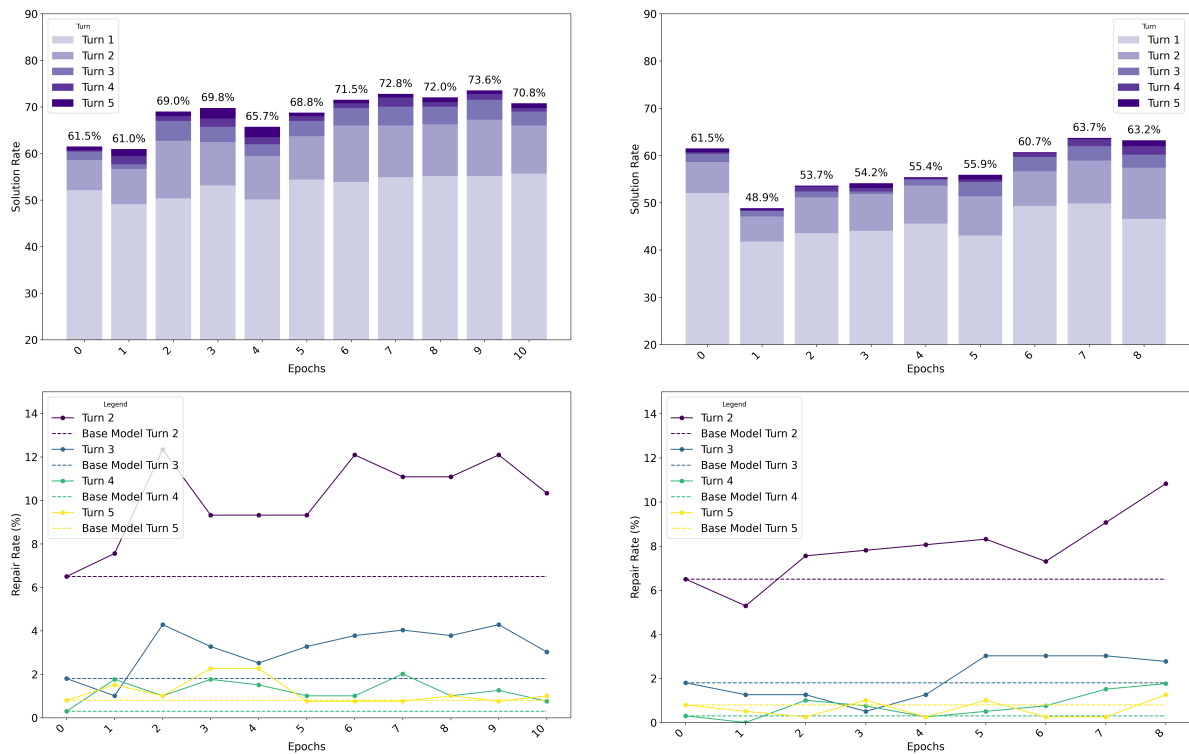


(b) Checkpoint scores and performance across turns of the *Qwen3-8B* model SFTed with **full fine-tuning** on *v1* dataset across 10 epochs in two regimes: thinking (left) and no-think (right).

Figure B.2: Comparison of LoRA (*v3* dataset) and full fine-tuning (*v1* dataset) across *thinking* and *no-thinking* regimes. The bar plots represent the scores of each individual checkpoint during training while the line plots represent the performance in subsequent turns.



(a) Checkpoint scores and performance across turns of the *Qwen3-8B* model SFTed with **full fine-tuning** on v3 dataset across 10 epochs in two regimes: thinking (left) and no-think (right).



(b) Checkpoint scores and performance across turns of the *Mistral-8B-Instruct-2410* model SFTed with LoRA (left) and **full fine-tuning** (right) on v5 dataset across 10 epochs.

Figure B.3: Comparison of full fine-tuning (v3 dataset) across *thinking* and *no-thinking* regimes for *Qwen3-8B* and the performance of *Mistral-8B-Instruct-2410* trained with LoRA and FFT. The bar plots represent the scores of each individual checkpoint during training while the line plots represent the performance in subsequent turns.

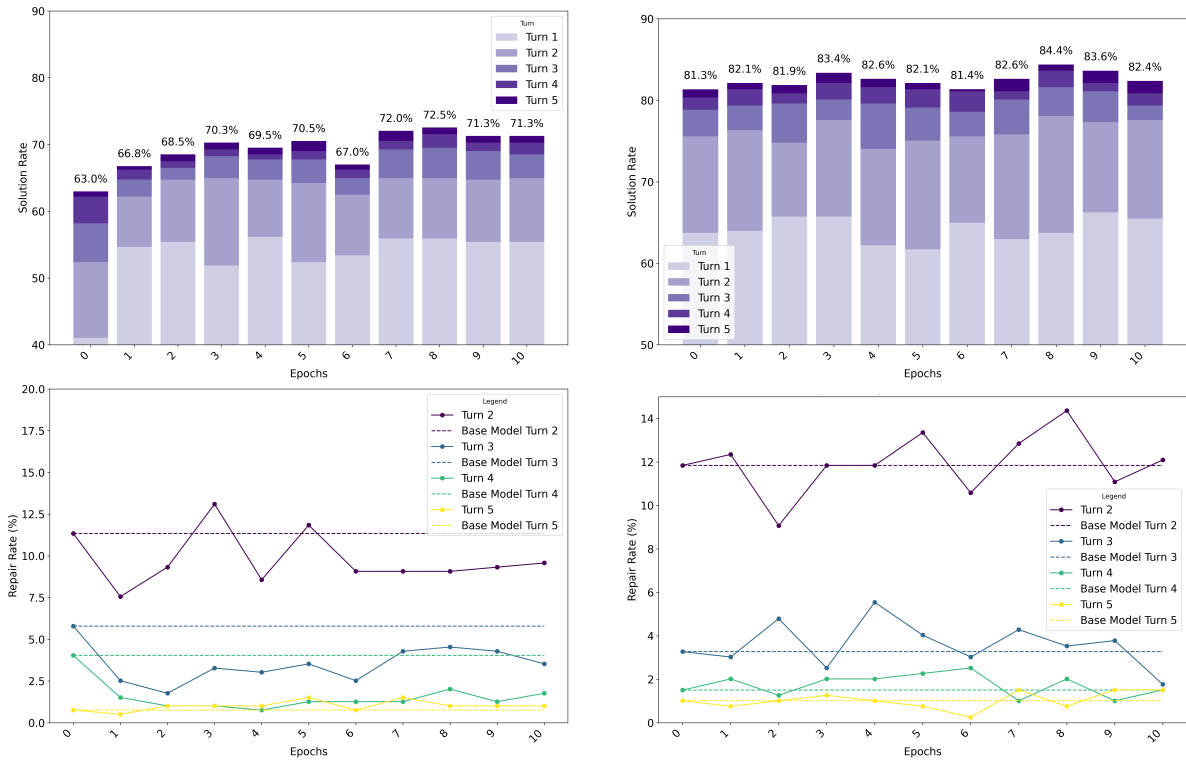


Figure B.4: Checkpoint scores and performance across turns of the *Llama-3.1-8B-Instruct* model SFTed with LoRA (left) and Qwen3-14B (right) on v5 dataset across 10 epochs.

B.2. The frequency of reasoning tokens across models.

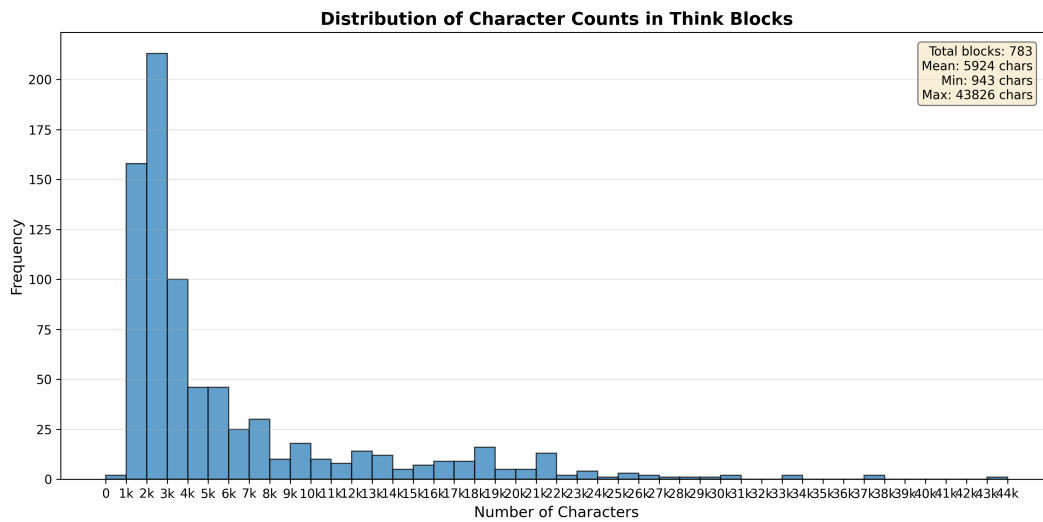


Figure B.5: The distribution of reasoning characters during evaluation on the Django benchmark of *Qwen3-8B* trained with reasoning distilled from *Qwen3-32B*.

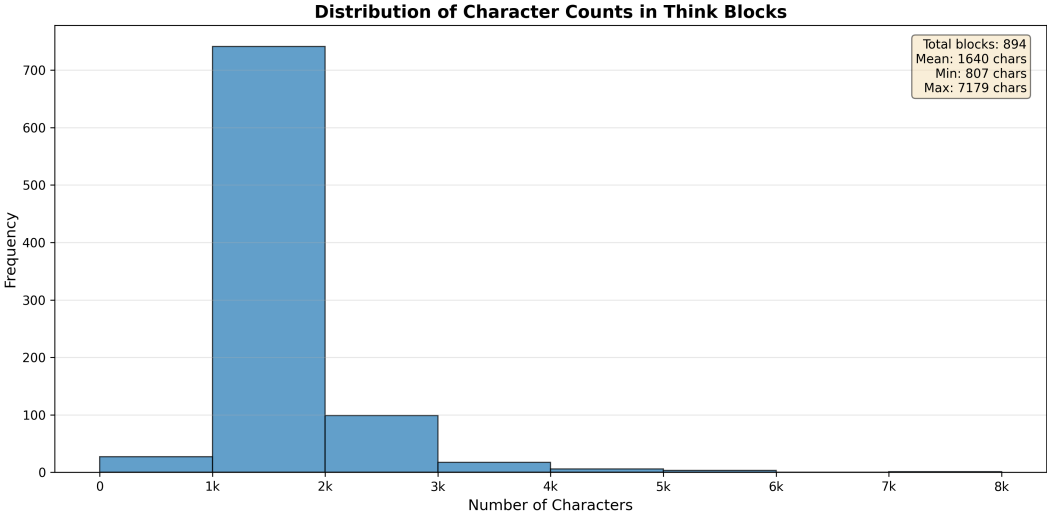


Figure B.6: The distribution of reasoning characters during evaluation on the Django benchmark of *Qwen3-8B* trained with shorter reasoning distilled from *Qwen3-32B*.

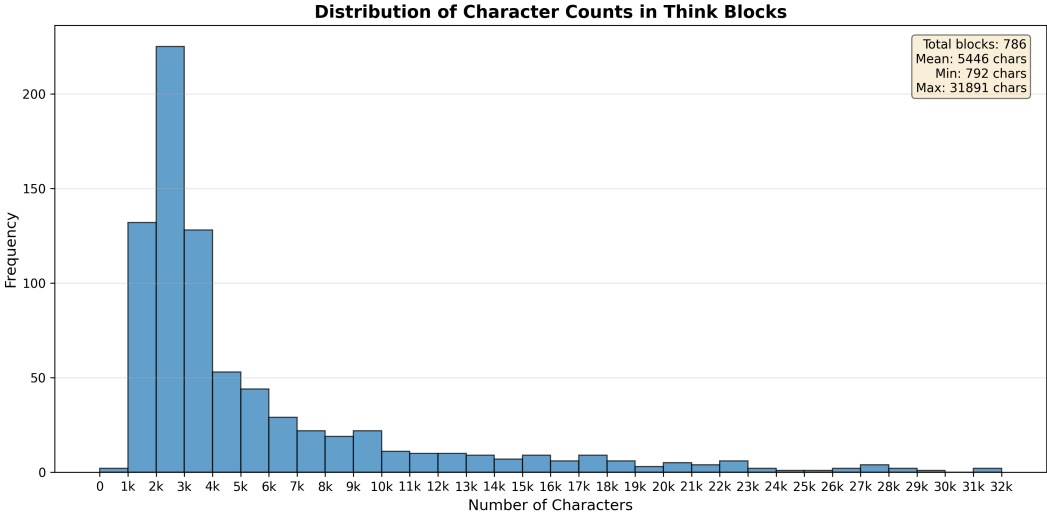


Figure B.7: The distribution of reasoning characters during evaluation on the Django benchmark of *Qwen3-8B* trained with reasoning distilled from *Qwen3-32B* and blended without the reasoning trajectories.

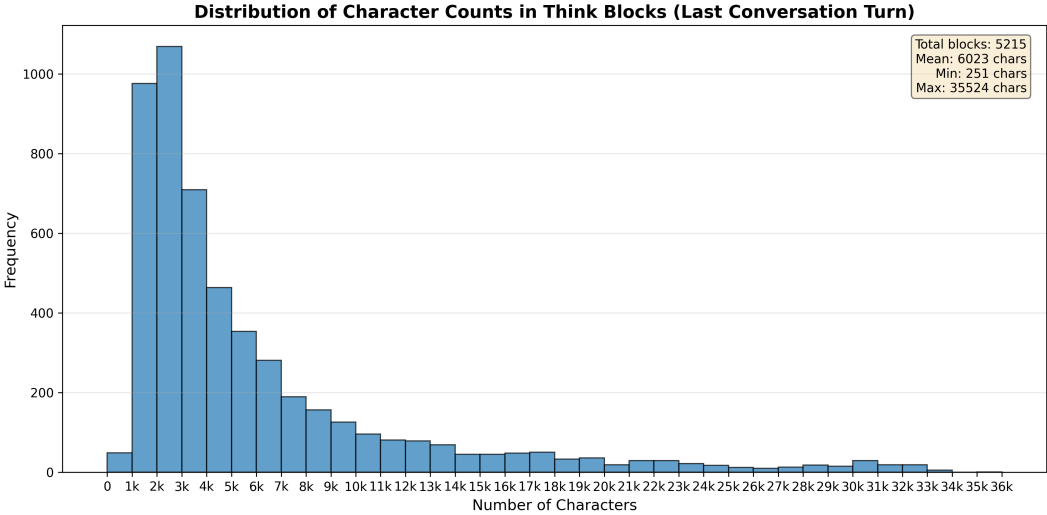
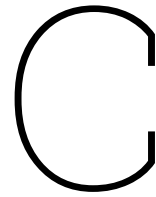


Figure B.8: The distribution of reasoning characters during evaluation on the Django benchmark of *Qwen3-8B* trained with reasoning distilled from *Qwen3-235B-A22B-Thinking-2507*.



Output Model examples

```
1 <think>Okay, let's see. The user is trying to fix the `get_all_permissions` method in the `
  PermissionsMixin` class. The tests are failing, so I need to figure out why. Looking at
  the previous attempts, the first version returned a list by combining user and group
  permissions with a union. But the test expected a set, and the output was a list. The
  second attempt used a set union, which fixed the type but didn't resolve all issues. The
  latest test failure shows that when an object is passed, the method is returning a list
  with 'none' and the expected is {'simple'}. Wait, the test is checking for `
  get_all_permissions(TestObj())`, which should return permissions specific to that object.
  But the current code is combining user and group permissions, which might not be object-
  specific. Wait, looking at the existing code in the `PermissionsMixin` class, the `
  get_user_permissions` and `get_group_permissions` methods are using `
  _user_get_permissions`, which takes `obj` as an argument. So when `obj` is provided,
  those methods return permissions specific to that object. However, the `
  get_all_permissions` method is supposed to combine both user and group permissions, but
  maybe the current approach isn't handling the object correctly. Wait, the test failure
  shows that when an object is passed, the method returns a list with 'none' and the
  expected is {'simple'}. That suggests that the method is not filtering permissions
  correctly based on the object. Maybe the `get_user_permissions` and `
  get_group_permissions` methods are not properly considering the object when it's provided
  . Wait, in the `get_all_permissions` method, the current code is using `self.
  get_user_permissions(obj) | self.get_group_permissions(obj)`. But if the object is
  provided, those methods should return permissions that are applicable to that object.
  However, the test is expecting only 'simple' but getting 'none' as well. That implies
  that the permissions are not being filtered correctly. Wait, maybe the `
  get_user_permissions` and `get_group_permissions` methods are returning all permissions,
  not just those applicable to the object. Let me check the code for `_user_get_permissions
  `. The function `_user_get_permissions` is called with `obj`, and it's supposed to return
  permissions that match the object. But if the backend's `get_user_permissions` method
  isn't filtering correctly, that could be the issue. Alternatively, perhaps the `
  get_all_permissions` method should call `_user_get_permissions` with "all" as the
  from_name, similar to how `AnonymousUser` does it. Wait, in the `AnonymousUser` class, `
  get_all_permissions` is defined as returning `_user_get_permissions(self, obj, "all")`.
  Maybe the `PermissionsMixin`'s `get_all_permissions` should be using the same approach,
  passing "all" to `_user_get_permissions` instead of combining user and group permissions.
  Wait, looking at the existing code in the `PermissionsMixin` class, the `
  get_all_permissions` method isn't defined. The `AnonymousUser` has a `get_all_permissions
  ` that uses `_user_get_permissions` with "all". So perhaps the correct approach is to
  call `_user_get_permissions` with "all" as the from_name, which would combine both user
  and group permissions through all backends. So the original code in the `PermissionsMixin
  `'s `get_all_permissions` should be using `_user_get_permissions` with "all" instead of
  combining user and group permissions. That would align with the `AnonymousUser`'s
  implementation. So the correct code would be: def get_all_permissions(self, obj=None):
  return _user_get_permissions(self, obj, "all"). But wait, the user's code in the `
  PermissionsMixin` doesn't have this method. The previous attempts tried to combine user
  and group permissions, but that's not the right approach. The correct way is to use the
  helper function `_user_get_permissions` with "all" as the from_name, which would handle
  both user and group permissions through all backends. So the correct code should be to
  call `_user_get_permissions` with "all" instead of combining the two. That would ensure
  that all permissions from both user and group are collected, and the object is considered
```

```

    if provided. So the final answer should be: def get_all_permissions(self, obj=None):
    return _user_get_permissions(self, obj, "all"). But wait, the user's code in the `
    PermissionsMixin` doesn't have this method. The `_user_get_permissions` function is
    defined in the code, so it's available. Therefore, the correct implementation is to use
    that helper function with "all" as the from_name.</think>
2
3
4 ```python
5 def get_all_permissions(self, obj=None):
6     return _user_get_permissions(self, obj, "all")

```

Listing C.1: Qwen3-8B reasoning trace used to repair previous failed test cases and resulted in pass@3. The reasoning of the model includes backtracking, verbosity and self-checking.

```

1 <think>Okay, let's see. The user is having issues with the get_all_permissions method in the
  PermissionsMixin class. The test is failing because when they call get_all_permissions on
  a user, it's returning 'none' instead of 'simple'. The previous attempts didn't account
  for the auth backends properly. Hmm, the original method was just combining user and
  group permissions. But the test failure suggests that the backend's get_all_permissions
  method isn't being called. Wait, the helper function _user_get_permissions is used in the
  User and AnonymousUser classes. Maybe the correct approach is to use that helper instead
  of directly calling get_user and get_group permissions. Looking at the code, the
  AnonymousUser's get_all_permissions uses _user_get_permissions with "all" as the
  from_name. The PermissionsMixin's method should probably do the same, because the current
  implementation might not be considering the backends correctly. So, the correct approach
  might be to call the _user_get_permissions function with "all" to get both user and
  group permissions through the backends. Also, check if the user is a superuser, but maybe
  the helper function already handles that. Let me check the _user_get_permissions
  function. The _user_get_permissions function loops through all backends and collects
  permissions. For the "all" case, it would collect both user and group permissions from
  each backend. So, the get_all_permissions method in PermissionsMixin should probably call
  this helper instead of combining the two separately. In the previous attempt, the code
  used _user_get_permissions when the user is a superuser, but otherwise combined user and
  group. But maybe the correct way is to always call the helper function. Let me think: the
  helper function for "all" would get all permissions from all backends. So the method
  should just return _user_get_permissions(self, obj, "all"). Wait, the test case is
  expecting the user to have the 'simple' permission. If the current code isn't using the
  backend's get_all_permissions, then it's not getting the correct permissions. So, the fix
  is to delegate to the helper function that queries the backends properly. So the correct
  method should be: def get_all_permissions(self, obj=None): return _user_get_permissions(
  self, obj, "all"). But wait, the previous code in the PermissionsMixin's
  get_all_permissions was combining user and group permissions, which might not include
  backend-specific logic. The helper function _user_get_permissions is designed to collect
  permissions from all backends. Therefore, replacing the manual combination with a call to
  the helper should fix the issue. Let me check the code again. The _user_get_permissions
  function is called with "user" or "group" as the from_name. For "all", it would collect
  both. Wait, no, the from_name is used to call get_user_permissions or
  get_group_permissions on each backend. But if the helper is called with "all", perhaps it
  's supposed to collect both user and group permissions. But how is that implemented?
  Looking at the _user_get_permissions function: the name is "get_%s_permissions" %
  from_name. So for "all", it would look for get_all_permissions in each backend. But the
  backends might not have that. Wait, in the test case, the RowlevelBackend probably has a
  get_all_permissions method that returns the correct permissions. Therefore, the correct
  approach is to use the helper function, passing "all" as the from_name. So the
  get_all_permissions method in PermissionsMixin should just return _user_get_permissions(
  self, obj, "all"). In the initial answer, the user tried combining user and group
  permissions, which did not go through the backends properly. The test failure suggests
  that the backend logic is not being applied. So the fix is to use the helper function
  that queries all backends for their get_all_permissions method.</think>
2
3 ```python
4 def get_all_permissions(self, obj=None):
5     return _user_get_permissions(self, obj, "all")

```

Listing C.2: Qwen3-32B reasoning trace for method repair, succeeded at pass@3. The reasoning is shorter than that of Qwen3-8B while still maintain a similar style of reasoning.

```

1 <think>Okay, I need to generate the get_all_permissions method for the PermissionsMixin class
  . Let me look at the existing code to understand what's going on. The PermissionsMixin
  has methods like get_user_permissions and get_group_permissions. The get_all_permissions
  is supposed to combine both user and group permissions. Looking at the AnonymousUser

```

```

class, its get_all_permissions method returns _user_get_permissions with "all". But in
the PermissionsMixin, the user and group permissions are handled by separate methods. The
_user_get_permissions function takes the user, obj, and a from_name (like "user", "group
", or "all"). The helper function is designed to collect permissions from all backends
based on the name. So for get_all_permissions, I should call _user_get_permissions with "
all" as the from_name. That should combine both user and group permissions. The method's
docstring should explain that it combines both and mentions the obj parameter. The
existing methods like get_user_permissions and get_group_permissions use this helper, so
following the same pattern here makes sense. I need to make sure the method is properly
declared with def get_all_permissions(self, obj=None): and returns the combined
permissions using the helper function. Also, adding a docstring as in the example would
be good for clarity.</think>
2
3 ```python
4 def get_all_permissions(self, obj=None):
5     """
6     Return a list of permission strings that this user has, both directly
7     and through groups. Query all available auth backends. If an object is
8     passed in, return only permissions matching this object.
9     """
10    return _user_get_permissions(self, obj, "all")

```

Listing C.3: The reasoning of the Qwen3-8B model trained on reasoning repair trajectories from Qwen3-32B, succeeding in generating the correct method at pass@1.

```

1 <think>Okay, I need to generate the get_all_permissions method for the PermissionsMixin class
. Let me look at the existing code to understand what's needed. The method should combine
user and group permissions through all auth backends. The existing methods
get_user_permissions and get_group_permissions use the _user_get_permissions helper,
which loops through backends and collects permissions. get_all_permissions should return
the union of these two sets. In PermissionsMixin, get_user_permissions calls
_user_get_permissions with "user", and get_group_permissions with "group". So
get_all_permissions should combine both results. The helper _user_get_permissions takes
the user, obj, and from_name ("user", "group", or "all"). When from_name is "all", it
should aggregate user and group permissions. Wait, but in the code, AnonymousUser's
get_all_permissions calls _user_get_permissions with "all". So maybe the helper handles
that when given "all". Looking at the helper's code: it loops through backends and
collects permissions based on from_name. So passing "all" to _user_get_permissions would
gather both user and group permissions. Therefore, the get_all_permissions method can
simply call _user_get_permissions with from_name="all". So the correct code is to return
_user_get_permissions(self, obj, "all"). That's concise and leverages existing logic to
combine user and group permissions across all backends.</think>
2
3 ```python
4 def get_all_permissions(self, obj=None):
5     return _user_get_permissions(self, obj, "all")

```

Listing C.4: The reasoning of the Qwen3-8B model trained on shortened reasoning repair trajectories from Qwen3-32B, succeeding in generating the correct method at pass@1.

```

1 <think>### Reasoning and Analysis of the Wrong Method The wrong method failed because it did
not properly remove the default `object` key from the context before adding the custom
key. Here's a breakdown of the issue: What the wrong method did: 1. It called `self.
get_context_data(object=self.object)`, which explicitly added the object to the context
under the default key. 2. Then it tried to rename the key by popping the old key and
inserting the new one. 3. However, this did not account for the possibility that the
default key was already in the context, and the pop operation was not guaranteed to
remove it. Why this failed the test: The test expected that the default key (`object`)
would not be present in the final context. But in the wrong method, the line removing the
default key only removed it if it existed under a specific name, and it did not remove
`object` itself. As a result, `object` remained in the context, which caused the test
to fail. --- ### How the Correct Solution Addresses the Problem The correct solution
ensures that the default key is explicitly removed from the context before adding the
custom key. It does this by: 1. Calling `self.get_context_data()` without passing `object
=self.object`, ensuring the default key is not added automatically. 2. Explicitly
removing both `object` and `context_object_name` from the context using `context.pop(
key, None)` to avoid errors and guarantee removal. 3. Adding the object to the new custom
key (`custom_{context_object_name}`). This guarantees that the default key is absent in
the final context. --- ### What to Be Careful About Next Time 1. ...</think>

```

Listing C.5: Reasoning example trace from GPT-5-mini highlighting the difference in tone, style and structure of reasoning compared to that of models from Qwen3 family.