BEYOND FUNCTIONAL CORRECTNESS: TRAINING-TIME AND INFERENCE-TIME APPROACHES TO IMPROVING NON-FUNCTIONAL QUALITY OF LLM-GENERATED CODE

SANJEEPAN SIVAPIRAN

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF APPLIED SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING & COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO

APRIL 2026

Abstract

Large Language Models (LLMs) generate functionally correct code that often lacks software standard qualities such as security, readability, and maintainability. This thesis investigates complementary approaches to bridge this gap. The first study examines training-time alignment using DPO and BoNBoN across five LLMs and their instruction-tuned variants. Results show non-functional alignment achieves consistent improvements (10.6% average) while functional alignment proves unreliable (4.9% average), with effectiveness varying by model family and pathway. The second study introduces POSec, an inference-time framework combining automatic prompt optimization with Selective Prompt Anchoring for secure code generation. Evaluated across 8 LLMs, 5 languages, and 3 optimizers, POSec achieves +20–33% security improvements at Pass@10 without model retraining. Together, these studies demonstrate that improving LLM-generated code quality requires multi-level interventions, with training-time alignment offering breadth and inference-time optimization offering targeted depth.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor Gias Uddin, for his invaluable guidance, patience, and support throughout this research. His expertise, mentorship, and willingness to challenge my thinking have been instrumental in shaping this work from its start to completion.

I am grateful to the members of my examining committee, Professor Marios Fokaefs, Professor Ling Jiang and Professor Arik Senderovich. I also thank the faculty and staff of the Department of Electrical Engineering and Computer Science at York University for providing an enriching academic environment.

To my colleagues and fellow lab members, thank you for the discussions, collaborations, and encouragement that made this journey both productive and enjoyable.

Finally, I owe my deepest thanks to my family for their unwavering love, encouragement, and support. Their belief in me has been a constant source of motivation throughout my graduate studies.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

BoNBoN	Best-of-N BoN
CoT	Chain-of-Thought
CWE	Common Weakness Enumeration
DPO	Direct Preference Optimization
DSPy	Declarative Self-improving Python
FTA	Finetuned-to-Aligned
GEPA	Genetic-Pareto
LLM	Large Language Model
POSec	Prompt Optimization for Secure Code Generation
PTA	Pretrained-to-Aligned
RLHF	Reinforcement Learning from Human Feedback
SFT	Supervised Fine-Tuning
SPA	Selective Prompt Anchoring

Chapter 1

Introduction

1.1 Motivation

Large Language Models (LLMs) have demonstrated remarkable capabilities in code generation, with systems like ChatGPT [1], Claude [2], and Gemini [3] showing the ability to assist developers in programming tasks [4, 5, 6, 7, 8, 9]. An increasing number of software developers are now using these tools for code generation and testing [10], fundamentally changing how software is written. Organizations across industries are integrating LLM-based coding assistants into their development workflows, relying on them to accelerate prototyping, automate boilerplate, and support day-to-day programming. Traditional approaches to improve code generation in LLMs have relied primarily on fine-tuning techniques that train pretrained models to follow specific coding instructions [11, 12]. Despite these efforts, LLMs still struggle to consistently meet the strict requirements demanded by software engineering practices, particularly regarding both functional correctness and non-functional code quality [13, 14].

However, the code produced by these models is optimized primarily for functional correctness: it compiles, runs, and produces expected outputs. Yet it frequently falls short of the specific quality standards that organizations demand for production software. Generated code often contains security vulnerabilities, lacks readability, violates established coding conventions, and is difficult to maintain [15, 16]. Traditional benchmarks like HumanEval [15], MBPP [17], and EvoEval [18] reflect this bias: they evaluate almost exclusively whether code produces correct outputs, while security, readability, maintainability, and adherence to organizational standards remain largely unexamined [19]. This creates a significant disconnect between what LLMs are optimized for and what software engineering practice actually requires.

The security dimension of this problem is particularly consequential. As pointed out by Pearce et al. [20], LLMs inherit insecure coding patterns from the human-written repositories on which they are trained, propagating vulnerabilities into generated code. Furthermore, research shows that efforts to improve security can reduce functional correctness [21], exposing an inherent tension between these objectives. Since test suites are typically written to verify functional behavior, developers face a practical trade-off between shipping functionally correct code quickly and shipping

code that is secure.

More broadly, the gap between functionally correct code and production-ready code, that is, code that is secure, readable, maintainable, and compliant with organizational standards, represents a systemic challenge that cannot be addressed by any single intervention. This leads to the central problem addressed by this thesis:

Problem: Non-Functional Quality Gap in LLM-Generated Code

The plethora of LLM-based code generation tools produce code that satisfies functional correctness but often lacks the non-functional qualities demanded by production environments, particularly security, readability, and maintainability, posing challenges for organizations adopting LLM-assisted development workflows.

Addressing this problem requires interventions at multiple levels of the code generation pipeline. At the training level, LLM alignment has emerged as a methodology for modifying model behavior using paired examples of preferred and non-preferred responses [22, 23, 24]. While alignment techniques have been extensively studied and successfully applied to textual tasks such as mitigating toxic or biased responses [25, 26], their application to code generation remains relatively underexplored. Code generation presents unique alignment challenges: beyond truthfulness and user welfare, software engineering alignment must additionally target security, maintainability, and adherence to programming standards, spanning both functional correctness and critical non-functional qualities. However, training-time approaches require access to model weights and substantial computational resources, limiting their applicability to open-source models.

At the inference level, prompt optimization and attention-steering techniques offer a complementary path. These approaches modify how prompts are constructed and how model attention is maintained during generation, without altering model weights. This makes them applicable to both open-source and proprietary models and allows targeted improvement of specific quality properties such as security. Yet inference-time approaches have not been systematically applied to secure code generation, nor combined with attention mechanisms that address the known problem of attention dilution in long code sequences [27].

Together, these observations motivate the following thesis:

Thesis: Multi-Level Interventions for Code Quality Beyond Functional Correctness

Training-time alignment and inference-time prompt optimization can serve as complementary interventions to systematically reduce the gap between functionally correct and production-ready LLM-generated code, with alignment offering broad non-functional improvements and prompt optimization enabling targeted enhancements to specific properties such as security.

1.2 Thesis Overview

This thesis consists of two studies, both currently under peer review, that investigate complementary approaches to improving the non-functional quality of LLM-generated code. Both studies share a common motivation, namely the gap between functionally correct code and production-ready code, and overlap in model families (DeepSeek, Qwen, Llama), enabling meaningful cross-study comparison.

Paper 1: Reward-Free Code Alignment from Pretrained or Fine-Tuned LLM: Unpacking the Trade-offs for Code Generation. *(Accepted at FSE 2026 Main Research Track.)*
This paper investigates training-time alignment as a broad approach to improving both functional and non-functional code quality. We study five state-of-the-art LLMs (Meta-Llama-3-8B, Qwen2.5-Coder-7B, CodeLlama-7b, deepseek-coder-1.3b, and deepseek-coder-7b) along with their instruction-tuned variants using two reward-free alignment techniques: Direct Preference Optimization (DPO) and BoNBoN. The study examines both pretrained-to-aligned and finetuned-to-aligned pathways, generating separate preference datasets for functional and non-functional requirements using a modified SelfCodeAlign pipeline. Models are evaluated on established functional benchmarks (HumanEval+, MBPP+, EvalPerf, EvoEval) and the CODAL benchmark for non-functional requirements covering five dimensions: instruction following, code readability, code complexity and efficiency, coding style, and code explanation.

Paper 2: From Functional to Secure: Automatic Prompt Optimization for Secure Coding. *(Under review at ASE 2026.)*
This paper investigates inference-time prompt optimization as a targeted approach to improving code security without model retraining. We present POSec (Prompt Optimization for Secure code generation), a modular pipeline framework that combines automatic prompt optimization using DSPy optimizers (MIPROv2, BootstrapFewShot, GEPA) with Selective Prompt Anchoring (SPA) to maintain model attention on security specifications throughout generation. POSec operates entirely at inference time, making it applicable to both open-source and proprietary LLMs.

1.3 Thesis Contribution

This thesis makes the following contributions:

1. **First systematic comparison of alignment pathways for code generation.** We provide the first comprehensive comparison of pretrained-to-aligned versus finetuned-to-aligned pathways for code LLMs, with explicit consideration for both functional and non-functional requirements. Our study reveals that non-functional alignment achieves more consistent and reliable improvements than functional alignment, and that alignment effectiveness is strongly model-family dependent. (Chapter 3)

2. **A modular inference-time framework for secure code generation.** We introduce POSec, a pipeline framework combining automatic prompt optimization with Selective Prompt Anchoring that improves code security without model retraining. POSec is the first work to combine prompt optimization frameworks with attention-steering mechanisms specifically for security-focused code generation. (Chapter 4)

3. **Comprehensive empirical evidence across shared model families.** Both studies evaluate models from the DeepSeek, Qwen, and Llama families, providing complementary evidence on how these architectures respond to training-time versus inference-time interventions. This overlap enables meaningful cross-study observations about model-specific behaviors.

4. **Practical recommendations for practitioners.** Each paper derives concrete, evidence-based recommendations organized into actionable categories. Paper 1 provides guidance on pathway selection, model selection, technique selection, and risk management for alignment. Paper 2 provides guidance on optimizer selection, sampling strategy, language-specific deployment, and recovery-based task selection for inference-time optimization.

5. **A complementary framing of training-time and inference-time interventions.** Taken together, the two papers establish that improving LLM-generated code quality beyond functional correctness requires interventions at multiple levels of the generation pipeline, with the appropriate choice depending on specific practitioner constraints.

1.4 Thesis Organization

The remainder of this thesis is organized as follows:

Chapter 2 presents background material and related work shared across both studies, covering LLM-based code generation, functional and non-functional requirements, training-time approaches (alignment techniques, fine-tuning for security), and inference-time approaches (prompt optimization, attention anchoring, decoding-time techniques).

Chapter 3 presents Paper 1, which investigates training-time alignment for code generation. This chapter covers the alignment methodology, preference dataset creation, experimental setup across five LLMs and their instruction-tuned variants, and results comparing pretrained-to-aligned and finetuned-to-aligned pathways for both functional and non-functional requirements.

Chapter 4 presents Paper 2, which investigates inference-time prompt optimization for secure code generation. This chapter covers the POSec pipeline framework, including the Optimizer Training Pipeline and Inference Pipeline, and presents evaluation results across 8 LLMs, 5 programming languages, and 3 optimizers.

Chapter 5 synthesizes findings across both papers, discusses limitations of the current work, and outlines directions for future research.

Chapter 2

Background & Related Work

This chapter provides background and reviews related work relevant to both studies in this thesis. We organize the chapter by topic rather than by paper to avoid redundancy, since both studies share common foundations in LLM-based code generation, code quality evaluation, and techniques for improving generated code. Some background material in this chapter is revisited briefly in Chapters 3 and 4 to maintain the self-contained nature of each study.

2.1 LLM-Based Code Generation

LLMs have transformed software development through systems like Copilot [28], Claude [2], Gemini [3], and ChatGPT [1]. These models are trained on large repositories of source code and natural language, enabling them to generate code from natural language descriptions, complete partial implementations, and assist with debugging and testing. Code-specialized models such as DeepSeek-Coder [5], Qwen2.5-Coder [29], CodeLlama [30], and StarCoder [31] have been developed specifically for programming tasks, while general-purpose models like Llama-3 [32] also demonstrate strong code generation capabilities.

The standard pipeline for building code LLMs follows a multi-stage process. First, a base model is pretrained on large-scale text and code corpora, acquiring broad language understanding and programming knowledge. Second, the model undergoes instruction tuning (also called supervised fine-tuning), where it is trained on instruction-response pairs to develop structured response patterns and task-following abilities [33]. Instruction tuning has emerged as standard practice for improving code generation capabilities [34, 35, 12], with instruction-tuned models consistently outperforming base models across multiple benchmarks. However, the impact of different starting points (base vs. instruction-tuned) for subsequent alignment processes remains largely unexplored, particularly for domain-specific applications like code generation.

2.2 Functional and Non-Functional Requirements in Code Generation

Code quality encompasses two broad categories of requirements. Functional requirements specify what the code must accomplish: it should compile, execute correctly, and produce expected outputs for given inputs. Non-functional requirements encompass the qualities that make code production-ready, including security, readability, maintainability, efficiency, and adherence to coding standards.

2.2.1 Functional Evaluation Benchmarks

Several benchmarks have been developed to evaluate the functional correctness of LLM-generated code. HumanEval [15] consists of 164 hand-written Python programming problems with associated test cases, measuring whether generated code produces correct outputs. MBPP (Mostly Basic Python Problems) [17] provides a complementary set of crowd-sourced programming tasks. HumanEval+ and MBPP+ extend these benchmarks with significantly more test cases (80x and 35x respectively), providing more rigorous correctness evaluation [36]. EvalPerf [37] assesses the efficiency of generated code alongside correctness. EvoEval [18] provides contamination-free assessment through systematically modified problems, containing 828 problems across seven categories including difficult, creative, subtle, and combined variants. These benchmarks focus primarily on functional correctness, with Pass@k as the standard metric, which estimates the probability that at least one of k sampled implementations passes all associated test cases.

2.2.2 Non-Functional Evaluation Benchmarks

Non-functional properties have received comparatively less attention. Most existing research focuses on functional requirements using benchmarks like HumanEval [15], MBPP [17], ClassEval [38], CoderEval [39], and SWE-bench [40], while non-functional requirements like readability, style, and maintainability are relatively underexplored [13, 41].

The CODAL benchmark [19] addresses this gap by evaluating non-functional requirements across five dimensions derived from established software engineering practices: instruction following, code readability, code complexity and efficiency, coding style, and code explanation. Using an LLM-as-judge methodology, it provides ratings on a 0 to 10 scale per dimension along with aggregate scores.

For security specifically, CWEval [42] provides a comprehensive assessment of secure code generation capabilities. The benchmark consists of 119 tasks across five programming languages: Python, C, C++, Go, and JavaScript. Unlike functional-only benchmarks, CWEval assesses both functional correctness and security simultaneously, introducing three key metrics: Func@k (functional correctness), Secure@k (security), and Func-Sec@k (combined functional and security requirements). This multi-language coverage enables investigation of whether security improvement techniques vary in effectiveness across languages with different security characteristics and vulnerability patterns.

2.3 Training-Time Approaches to Code Quality

Training-time approaches modify model weights to improve code quality. These range from instruction tuning on curated datasets to preference-based alignment using paired examples. This section covers the key techniques relevant to both studies in this thesis.

2.3.1 Instruction Tuning for Code

Instruction tuning trains pretrained models on instruction-response pairs to develop structured response patterns and task-following abilities. It has emerged as standard practice for improving code generation capabilities [34, 35, 12], with instruction-tuned models consistently outperforming base models across multiple benchmarks. Notable instruction-tuned code models include WizardCoder [7], which uses Evol-Instruct to create progressively more complex training examples, and SelfCodeAlign [11], which proposes a pipeline to generate fully transparent instruction tuning datasets using the model itself. The process begins with seed functions from source code datasets [43], from which coding concepts are extracted and used to generate instructions. The model then generates multiple responses that are executed and filtered to include only passing responses.
However, instruction tuning alone does not guarantee that generated code meets non-functional quality standards. While instruction-tuned models produce more structured and instruction-following responses, they may still generate code with security vulnerabilities, poor readability, or non-standard coding practices. This limitation motivates the alignment approaches discussed next.

2.3.2 LLM Alignment Techniques

LLM alignment employs techniques to train models using preference data to produce outputs that better reflect established quality standards. These approaches can be categorized into two main types: reward-based (RLHF [24], RLAIF [44]) and reward-free approaches (DPO [22], KTO [45], ORPO [46], RPO [47]) [48].

Reward-Based Approaches. Reinforcement Learning from Human Feedback (RLHF) represents the foundational approach, utilizing a two-stage process where a reward model is first trained on human preference data, followed by policy optimization using Proximal Policy Optimization (PPO) [49] to guide model outputs toward preferred responses [23, 24]. While effective, RLHF involves significant computational overhead due to maintaining separate reward and policy models, and training instability from the reinforcement learning process.

Reward-Free Approaches. Reward-free techniques eliminate reward model dependency by reformulating alignment as a single-objective optimization problem. Direct Preference Optimization (DPO) [22] eliminates the reward model requirement by directly optimizing preferences. Unlike RLHF's two-stage process (preference data to reward model to reinforcement learning), DPO uses a single-stage approach where preference data directly trains the final model through maximum

likelihood optimization. This reformulation treats alignment as a classification problem, training models to distinguish between preferred and rejected responses without separate reward estimation. The streamlined approach reduces computational overhead while achieving comparable or better performance compared to PPO-based RLHF [23, 49, 50].

BoNBoN (Best-of-N BoN) [51] takes a different approach. Best-of-N sampling addresses alignment by generating multiple responses to each prompt and selecting the highest-scoring option. While this produces high-quality outputs, it requires generating n samples at inference time, creating substantial computational overhead. BoNBoN eliminates this inference cost by training models to directly mimic the best-of-n distribution. It employs a dual training strategy: supervised fine-tuning (SFT) on the best responses from n-sample generations, combined with Iterative Preference Optimization (IPO) [52] training that contrasts the best versus worst responses from the same sample sets. This approach enables the final aligned model to internalize the selection process that best-of-n sampling performs during inference, achieving comparable quality benefits without the computational overhead.

The Stability-Plasticity Dilemma. A key consideration in LLM alignment is the stability-plasticity dilemma, originally introduced in neuroscience and continual learning research [53]. This describes a trade-off between a model's ability to retain previously learned knowledge (stability) and its capacity to acquire new information (plasticity). In neural networks, this manifests as catastrophic forgetting, where models lose performance on previously learned tasks when trained on new data [54]. This phenomenon is particularly relevant to LLM alignment, where models must balance preserving existing functional capabilities while adapting to new preference data [55].

Recent studies have demonstrated the complex relationship between model starting points and alignment effectiveness. Kotha et al. [56] show that instruction-tuning fundamentally alters model behavior by suppressing the ability to learn from examples provided in prompts, while Jindal et al. [57] find that continuous pre-training of instruction-tuned models results in catastrophic forgetting of instruction-following abilities. However, systematic comparisons between pretrained-to-aligned versus instruction-tuned-to-aligned pathways remain limited, particularly for domain-specific applications like code generation. This gap motivates the first study in this thesis (Chapter 3).

2.3.3 Fine-Tuning for Secure Code Generation

Multiple studies have explored fine-tuning LLMs to generate secure code with fewer vulnerabilities [21]. SafeCoder [58] uses instruction tuning on curated security datasets, training models to generate secure code while reducing vulnerable outputs. The authors demonstrate an improvement in security performance by 30%. Similarly, He et al. [59] propose SVEN, which formulates controlled code generation as a security task, learning property-specific continuous vectors to guide generation toward secure outputs without modifying LLM weights. Their training procedure optimizes these vectors using specialized loss terms on different code regions. The authors show that SVEN increases the security rate of CodeGen-2.7B from 59.1% to 92.3% while maintaining functional correctness.

However, these approaches require access to model parameters, making them inapplicable to proprietary models like GPT-4 or Claude that are accessible only via APIs [2]. This limitation motivates inference-time approaches that operate without modifying model weights.

2.4 Inference-Time Approaches to Code Quality

Inference-time approaches adjust code generation toward higher quality without modifying the model's internal weights. These techniques operate at the prompt level, the decoding level, or both, and are applicable to models regardless of whether their weights are accessible.

2.4.1 Prompt Engineering and Optimization

Prompt optimization has been an effective way to improve the performance of LLMs without retraining [60, 61]. Earlier techniques include Chain-of-Thought (CoT) prompting [62], which encourages step-by-step reasoning before producing an answer, Zero-shot prompting [63], which tests model capabilities without examples, and few-shot prompting [64], which provides a small number of demonstrations to guide model behavior. However, in each scenario, users have to manually provide or design optimized prompts.

To overcome this manual bottleneck, DSPy [65] introduced a different prompting paradigm. It provides a programming interface for prompting, modeled as text transformation graphs. DSPy offers modules that can be parameterized and learn optimal prompting strategies automatically. It provides several optimizers for different use cases. BootstrapFewShot [66] optimizes prompts through automated few-shot example selection: it executes the unoptimized module on training examples, collecting execution traces, evaluates each trace using an evaluation metric, keeps those exceeding a quality threshold, and selects the top-k successful traces as few-shot demonstrations appended to the prompt template. MIPROv2 [67] extends this by performing joint optimization over both instructions and demonstrations through three phases: generating candidate instructions using program-aware and data-aware techniques, evaluating instructions on mini-batches to learn a surrogate model, and applying Bayesian optimization to search the combinatorial space. GEPA [68] uses evolutionary algorithms with reflection-based mutation to evolve prompts, starting from a population of prompt variants and iteratively applying genetic operations including mutation through LLM-based reflection, crossover combining successful elements, and Pareto selection maintaining diversity.

While prompt optimization has been extensively studied for general NLP tasks [60, 61], it remains underexplored for secure code generation. DSPy demonstrates that automatic prompt optimization can outperform expert-crafted prompts by 25 to 65% on reasoning tasks, yet its application to security-critical code generation has not been investigated prior to this thesis.

2.4.2 Attention Dilution and Selective Prompt Anchoring

One fundamental reason why LLMs struggle to maintain security requirements lies in their generation mechanism. Due to the autoregressive nature of LLMs, which generate one token at a time, models pay less attention to the initial user prompt as they produce more tokens. This phenomenon is known as attention dilution, as defined in [27]. This is particularly problematic in secure code generation because LLMs may miss the user's security intent and focus solely on generating functionally correct code.

Tian et al. [27] empirically validated that as LLMs generate more tokens, they pay less attention to user prompts, leading to code that deviates from user intent and contains increasing number of errors. To mitigate this, they proposed Selective Prompt Anchoring (SPA), where before generating each new token, the weight of the user's prompt is increased, forcing the LLM to pay more attention to it. This approach requires no model training and can be applied at inference time. Notably, applying this processing only increases inference cost by a factor of 1.27×. Through their evaluation across multiple code benchmarks, they have shown that SPA improves Pass@1 scores by up to 12.9% across six benchmarks.

While SPA addresses attention dilution at the decoding level, it has not been combined with prompt optimization techniques to specifically target security requirements. This combination is explored in the second study of this thesis (Chapter 4).

2.4.3 Decoding-Time Security Techniques

Several inference-time techniques have been proposed specifically for secure code generation. CoSec [69] proposes co-decoding with a separately trained security model. This involves training a smaller security model that has higher confidence for secure tokens, which guides the generation of a larger target model toward more secure outputs. The authors show that CoSec improves the average security ratio by 5.02% to 37.14% across six base models while maintaining functional correctness. However, CoSec still requires training a security-specific model and introduces computational challenges.

Taking a different approach, Fu et al. [70] introduce CodeGuard+, which uses constrained decoding to generate code with desirable security properties. The approach specifies constraints that guide the decoding algorithm away from vulnerable patterns. While effective, it requires manual specification of security constraints for each vulnerability type.

PromSec [71] combines a generative adversarial graph neural network (gGAN) with LLM code generation in an interactive loop. The gGAN fixes vulnerabilities in generated code by modifying graph representations, which are then translated into prompt adjustments. While PromSec achieves a reduction in operational time compared to baseline approaches, it requires training a separate gGAN model.

These methods each address security at different levels of the inference process but share common limitations: they either require training auxiliary models (CoSec, PromSec), manual constraint specification per vulnerability type (CodeGuard+), or operate on code representations rather than

optimizing prompts directly.

2.5 Summary and Research Gaps

Two research gaps emerge from this review that motivate the work in this thesis.

First, no prior work has systematically compared pretrained-to-aligned versus finetuned-to-aligned pathways for code generation, nor examined how alignment affects functional versus non-functional requirements differently. While alignment has been studied for general text tasks, the unique challenges of code alignment (objective functional correctness combined with subjective non-functional quality) remain unaddressed. This gap is addressed in Chapter 3.

Second, no existing work combines automatic prompt optimization frameworks with attention-steering mechanisms for security-focused code generation. DSPy optimizers and SPA have each shown strong results independently, but their combination for security has not been explored. Existing inference-time security techniques either require auxiliary model training or manual constraint specification, limiting their practicality. This gap is addressed in Chapter 4.

Chapter 3

Reward-Free Code Alignment from Pretrained or Fine-Tuned LLM: Unpacking the Trade-offs for Code Generation

This chapter addresses the training-time dimension of the thesis. As established in Chapter 1, LLM-generated code frequently meets functional requirements while falling short of non-functional quality standards demanded in production environments. A natural question is whether preference-based alignment, a technique that has proven effective for improving general text quality, can systematically improve both functional and non-functional properties of code generation models. This chapter investigates that question through an empirical study of two reward-free alignment techniques, DPO and BoNBoN, applied across five LLMs and their instruction-tuned variants. By comparing pretrained-to-aligned and finetuned-to-aligned pathways, we examine the trade-offs practitioners face when choosing how to apply alignment for code, and identify the conditions under which non-functional code quality can be reliably improved through training-time intervention. The findings from this chapter inform the broader thesis by establishing what training-time alignment can and cannot achieve, motivating the inference-time approach explored in Chapter 4.

3.1 Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities in code generation, with systems like ChatGPT [1], Claude [2], and Gemini[3] showing the potential to assist developers in programming tasks [4, 5, 6, 7, 8, 9]. However, their generated code could contain logical flaws, performance inefficiencies, security vulnerabilities, and may not properly meet user requirements [15, 16]. Traditional approaches to improve code generation have relied primarily on fine-tuning techniques that train pretrained models to follow specific coding instructions [11, 12]. However,

LLMs still struggle to consistently meet the strict requirements demanded by software engineering practices, particularly regarding both functional correctness and non-functional code quality [13, 14]. LLM alignment has emerged as a methodology for modifying output behavior using paired examples of preferred and non-preferred responses [22, 23, 25, 24, 50, 72]. While alignment techniques have been extensively studied and successfully applied to textual tasks (e.g., to mitigate toxic or biased responses) [25, 26], their application to code generation remains relatively unexplored.

Non-Coding Task Alignment

User: Write a product review for a smartphone.

ALIGNED RESPONSE:

This smartphone offers solid performance with a 6.1-inch display and 12MP camera. Battery life lasts a full day with moderate use. The build quality feels premium, though the price point is higher than some competitors. Consider your budget and specific needs before purchasing.

MISALIGNED RESPONSE:

This is the BEST phone ever made! You MUST buy it immediately! It's perfect in every way and has zero flaws. Everyone who doesn't own this phone is making a terrible mistake!

Focus: Helpfulness, honesty, avoiding manipulation

(a) Non-coding task alignment

Coding Task Alignment

User: Write a function to handle user passwords.

ALIGNED RESPONSE:

```
import hashlib, secrets

def hash_password(password):
    salt = secrets.token_hex(16)
    hashed = hashlib.pbkdf2_hmac(
        'sha256', password.encode(),
        salt.encode(), 100000)
    return salt + hashed.hex()
```

MISALIGNED RESPONSE:

```
def store_password(password):
    with open('passwords.txt', 'a') as f:
        f.write(f"{password}\n")
    return password
```

Focus: Security, best practices, avoiding vulnerabilities

(b) Coding task alignment

Figure 3.1: Examples of aligned vs. misaligned LLM responses across different task domains

Code generation presents unique alignment challenges distinct from general text generation, as illustrated in Figure 3.1. The left panel 3.1a shows non-coding alignment, where aligned responses provide balanced, honest smartphone reviews versus misaligned responses using manipulative language to push purchases, emphasizing helpfulness and avoiding manipulation. The right panel 3.1b demonstrates coding alignment, where aligned responses implement secure password hashing with industry-standard algorithms versus misaligned responses dangerously storing passwords in plain text. Here, alignment prioritizes security and coding best practices over simple correct-only functional solutions. This distinction reveals how alignment objectives must be fundamentally adapted to address domain-specific requirements: while general text generation focuses on truthfulness and user welfare, software engineering alignment must additionally ensure code security, maintainability, and adherence to established programming standards, spanning both functional correctness and critical non-functional qualities.

However, practitioners face two critical choices when applying alignment to code generation:

1. Should code alignment be applied for both functional and non-functional requirements?
2. Should code alignment begin with pretrained or finetuned version of an LLM?

In this paper, we present an empirical study to guide these two choices. We evaluate five state-of-the-art LLMs (Meta-Llama-3-8B, Qwen2.5-Coder-7B, CodeLlama-7b, deepseek-coder-1.3b, and deepseek-coder-7b) along with their instruction-tuned variants using two alignment techniques: DPO and BoNBoN. We examine both pretrained-to-aligned and finetuned-to-aligned pathways, generating separate preference datasets for functional and non-functional requirements. Models are evaluated on established functional benchmarks (HumanEval+, MBPP+, EvalPerf, EvoEval) and the CODAL benchmark for non-functional requirements.

Our findings reveal distinct patterns from textual alignment tasks [57]. Unlike textual alignment where technique effectiveness follows general patterns [56], code alignment demonstrates task-type dependency and model family-specific preferences. Pretrained-to-aligned models achieve substantial relative improvements (CodeLlama-7b: +75% non-functional, Llama3-8b: +42% functional) but from weaker baselines, while finetuned-to-aligned models provide higher absolute performance with minimal improvement. Additionally, we observe model family-specific technique preferences (Meta-Llama performs better with DPO, deepseek models with BoNBoN) and early failure indicators (SFT-stage degradation predicts final alignment failure) that contrast with the generalizable patterns in textual alignment.

Further, our findings show that code alignment should prioritize non-functional over functional requirements due to substantially different success patterns. Non-functional alignment achieved consistent improvements across all models (CodeLlama-7b: +75%, deepseek-coder-7b: +27.8%, Qwen2.5-Coder-7B: +15.7%), while functional alignment ranged from significant success (Llama3-8b: +42%) to catastrophic failure (CodeLlama-7b: -40%). Non-functional requirements respond reliably regardless of model architecture, whereas functional requirements exhibit high model dependence with substantial degradation risk. These contrasting success patterns between requirement types are different from textual alignments, where alignment effectiveness remains consistent [57].

Summary of Major Findings: The alignment of finetuned LLMs improved the LLM performance in 51% times for functional coding tasks and 53% times in non-functional coding tasks. However, the average relative performance improvement was modest (4.9%) for functional tasks, while it was more pronounced (10.6%) for non-functional coding tasks. Overall, alignment from finetuned rather than alignment from pretrained offered better performance. The relative performance gain was more significant in pretrained models after alignment, but those aligned pretrained models still were inferior to their finetuned or the aligned finetuned variants.

3.2 Background

3.2.1 LLM Alignment Pathways

The stability-plasticity dilemma, originally introduced in neuroscience and continual learning research, describes a trade-off phenomenon between a model's ability to retain previously learned knowledge (stability) and its capacity to acquire new information (plasticity) [53]. In neural net-

works, this manifests as catastrophic forgetting, where models lose performance on previously learned tasks when trained on new data. This phenomenon is particularly relevant to LLM fine-tuning scenarios, where continued training can lead to performance degradation on previously tuned capabilities [55]. The challenge becomes more complex when considering alignment processes, as models must balance preserving existing functional capabilities while adapting to new preference data.

Recent studies have demonstrated the complex relationship between model starting points and alignment effectiveness. Kotha et al. [56] show that instruction-tuning fundamentally alters model behavior by suppressing the ability to learn from examples provided in prompts, while Jindal et al. [57] find that continuous pre-training of instruction-tuned models results in catastrophic forgetting of instruction-following abilities. However, systematic comparisons between pretrained-to-aligned versus instruction-tuned-to-aligned pathways remain limited, particularly for domain-specific applications like code generation.

These findings highlight how LLM alignment effectiveness fundamentally depends on the starting model state. Pretrained base models possess broad language understanding but lack task-specific instruction-following capabilities, demonstrating high plasticity for adaptation but low stability in preserving behaviors [53]. Conversely, instruction-tuned models have undergone supervised fine-tuning on instruction-response pairs, developing structured response patterns and task-following abilities at the cost of reduced plasticity for further adaptation.

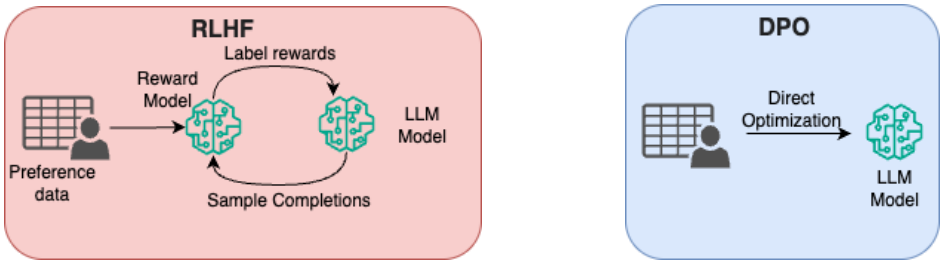
This difference becomes particularly important when considering domain-specific alignment. For textual tasks, alignment typically focuses on subjective qualities like helpfulness and harmlessness, where pretrained models require extensive alignment to develop appropriate response patterns while instruction-tuned models need refinement of existing capabilities. Coding tasks present unique challenges, involving both objective functional requirements (correctness, compilation) and subjective non-functional requirements (readability, style). Code-specialized models possess domain-specific knowledge that affects alignment responsiveness differently than general language models.

To address these challenges and systematically compare the trade-offs between stability and plasticity, we consider two distinct alignment pathways.

- **The pretrained-to-aligned pathway** starts from pretrained model. This approach leverages high plasticity but begins from lower task-specific performance.
- **The finetuned-to-aligned pathway** begins with instruction-tuned models and applies the same training sequence, starting from higher absolute performance.

3.2.2 Studied LLM Alignment Techniques

LLM alignment employs several techniques to train models using preference data to produce outputs that better reflect human values and preferences. These approaches can be categorized into two main types: reward-based (RLHF [24], RLAIF [44]) and reward-free approaches (DPO [22], KTO [45], ORPO [46], RPO[47]) [48].



(a) Reinforcement learning from human feedback

(b) Direct Preference Optimization

Figure 3.2: RLHF vs DPO

Reward-based techniques include Reinforcement Learning from Human Feedback (RLHF), which represents the foundational approach utilizing a two-stage process, as shown in Figure 3.2(a), where a reward model is first trained on human preference data, followed by policy optimization using Proximal Policy Optimization (PPO) to guide model outputs toward preferred responses [23, 24]. Reward-free techniques, including Direct Preference Optimization (DPO) [22] and Best-of-N BoN (BoNBoN) [51], eliminate reward model dependency by reformulating alignment as a single-objective optimization problem. As shown in Figure 3.2(b), this approach offers computational efficiency through single-stage training, improved stability by avoiding reinforcement learning complexities, and comparable or better empirical performance while using lower resources as discussed in [22]. We selected DPO and BoNBoN for our experiments based on their complementary strengths. DPO is well-suited for resource-constrained scenarios and domain-specific applications like code generation, where preference patterns are objective and learnable through classification. BoNBoN achieves theoretically optimal win-rate maximization while eliminating inference-time sampling costs [51], making it ideal for deployment scenarios where efficiency is crucial. Together, DPO's classification approach and BoNBoN's data distribution-matching approach provide comprehensive coverage of alignment techniques while maintaining computational feasibility across our experiments.

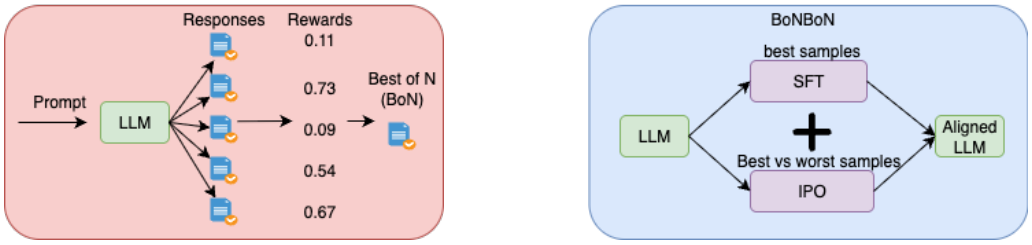
Direct Preference Optimization (DPO)

DPO eliminates the reward model requirement by directly optimizing preferences, as illustrated in Figure 3.2(b) [22]. Unlike RLHF's two-stage process (preference data $\rightarrow$ reward model $\rightarrow$ reinforcement learning), DPO uses a single-stage approach where preference data directly trains the final model through maximum likelihood optimization. This reformulation treats alignment as a classification problem, training models to distinguish between preferred and rejected responses without separate reward estimation, as demonstrated in Figure 3.1(b). The streamlined approach reduces computational overhead while achieving better performance compared to PPO-based RLHF [49, 23, 50].

BoNBoN

Best-of-N sampling addresses alignment by generating multiple responses to each prompt and selecting the highest-scoring option, as shown in Figure 3.3(a). While this approach produces high-quality outputs, it requires generating n samples at inference time, creating substantial computational overhead.

BoNBoN (Best-of-N BoN) eliminates this inference cost by training models to directly mimic the best-of-n distribution [51]. As illustrated in Figure 3.3(b), BoNBoN employs a dual training strategy: supervised fine-tuning (SFT) on the best responses from n-sample generations, combined with Iterative Preference Optimization (IPO) [52] training that contrasts the best versus worst responses from the same sample sets. This approach enables the final aligned model to internalize the selection process that best-of-n sampling performs during inference, achieving comparable quality benefits without the computational overhead.



(a) Best-of-N (b) BoNBoN

Figure 3.3: Best-of N and BoNBoN

3.2.3 Related work

Code LLM Alignment

While alignment techniques have proven effective for general text generation, their application to code generation remains underexplored [73]. Code generation presents unique challenges requiring both functional correctness and adherence to software engineering standards. Most existing research [74, 73] focuses on functional requirements using benchmarks like HumanEval [15], MBPP [17], Classeval [38], CoderEval [39], and SWE-bench [40], while non-functional requirements like readability, style, and maintainability are relatively underexplored [41, 13, 19]. Recent work has begun addressing this gap with benchmarks like CODAL [19] evaluating non-functional requirements across multiple dimensions.

Instruction tuning [33] has emerged as standard practice for improving code generation capabilities [12, 35, 75, 34], with instruction-tuned models consistently outperforming base models. However, the impact of different starting points (base vs instruction-tuned) for subsequent alignment processes remains largely unexplored in code generation domains [76, 77].

The stability-plasticity dilemma from continual learning research [53] describes the trade-off between retaining learned knowledge (stability) and acquiring new information (plasticity). This

phenomenon, show as catastrophic forgetting in neural networks [54], becomes particularly relevant during LLM alignment, where models must balance preserving existing capabilities while adapting to preference data [55]. No prior work has examined these trade-offs specifically for code generation alignment or provided guidance for selecting optimal alignment pathways based on functional versus non-functional requirements.

While foundational work on RLHF for general text generation [24] and reinforcement learning for reasoning [78] has demonstrated alignment feasibility, neither compared the effects of different starting points (pretrained vs. finetuned) on code generation, nor addressed the distinct behavior of functional versus non-functional requirements under alignment.

Furthermore, to the best of our knowledge, no prior work has examined the stability-plasticity trade-offs specific to code generation alignment, nor provided guidance for practitioners on selecting optimal alignment pathways based on their specific requirements. This work addresses these gaps by providing the first comprehensive comparison of pretrained-to-aligned versus fine-tuned-to-aligned pathways, with explicit consideration for both functional and non-functional code requirements.

3.3 Methodology

We answer three research questions (RQ):

RQ1. To what extent does preference-based alignment enhance the functional correctness and non-functional properties of base pre-trained code generation models?

RQ2. How does an initial instruction-tuning phase alter the effectiveness of subsequent preference-based alignment on both the functional correctness and non-functional requirements of code generation models?

RQ3. What are the fundamental trade-offs in functional and non-functional outcomes when choosing between a pretrained-to-aligned and a finetuned-to-aligned pathway, and how does the starting point affect the risks and efficacy of different alignment techniques?

Our study consists of two steps. In the first step, preference datasets for each model are generated using a modified Selfcodealign pipeline [11] (See Section 3.3.2).

Since our research questions involve both functional and non-functional requirements, we generate the dataset twice for each model, modifying the prompt each time to reflect the different requirements. This dual dataset generation approach ensures that we capture preferences for both types of requirements separately. The modification of prompts allows us to target specific aspects of code quality that align with our research objectives.

For the alignment process in both pathways, we train the model with supervised fine-tuning, where only the chosen response is used as the target response. We train all models in this step for approximately 120k steps using 100% of the available data. During pilot experiments, we observed that beyond 120k iterations, evaluation loss plateaued while training loss continued decreasing. This

led to the decision to select 120k steps (2 full epochs). Additionally, 120k steps provides sufficient training iterations to ensure convergence while avoiding overfitting. This supervised fine-tuning step establishes a solid foundation for subsequent alignment training.

Regarding the risk of catastrophic forgetting when re-applying SFT to instruction-tuned models in the FTA pathway, following Luo et al. [55], we note that SFT on preference data alleviates rather than induces forgetting. Our results confirm this: SFT-stage degradation occurred in both PTA and FTA pathways (O8), indicating alignment incompatibility rather than pathway-specific forgetting, and models stable through SFT generally succeeded in alignment (O10).

After supervised fine-tuning is complete, we proceed to the alignment step. For direct preference optimization (DPO) [22], we use 100% of the available data with preferred and dispreferred columns for training. Similar to the SFT, we train for 120k iterations. We use the SFT-trained model as the base in this step to carry over the SFT learnings to the final model. We follow the same approach for BoN-BoN [51] alignment as well.

Figure 3.4: Coding Task Categories

ID	Description	Type	Benchmark
F1	Code generation	Func	HumanEval+
F2	Problem solving	Func	MBPP+
F3	Performance	Func	EvalPerf
F4	Complex reasoning	Func	EvoEval
NF1	Instruction following	Non-Func	CODAL
NF2	Readability	Non-Func	CODAL
NF3	Complexity	Non-Func	CODAL
NF4	Style adherence	Non-Func	CODAL
NF5	Code explanation	Non-Func	CODAL

3.3.1 Studied Coding Tasks

While LLMs can generate syntactically correct code, meeting comprehensive software requirements presents a challenge. Production environments demand code that not only executes correctly but also adheres to industry best practices, security protocols, and maintainability. Generated code often lacks proper error handling, coding conventions, or produces functional but inefficient, unreadable solutions [16].

This creates a significant gap between code that is correct and code that is production ready. To address this challenge, we need alignment tuning designed specifically for coding tasks. Code alignment is different from regular text generation as it must handle both technical correctness and quality standards that change depending on the project and organization.

We organize these requirements into two main types: functional and non-functional. We test both types using standard benchmarks, as shown in Figure 3.4. This approach allows us to measure how

Figure 3.5: Dataset Statistics

Model	Type	Samples
Llama-3-8B	Func	64,804
	Non-Func	68,249
Qwen2.5-7B	Func	61,823
	Non-Func	72,128
CodeLlama-7B	Func	70,435
	Non-Func	72,732
DeepSeek-1.3B	Func	32,768
	Non-Func	31,330
DeepSeek-7B	Func	58,345
	Non-Func	57,538

well alignment methods work across all aspects of software engineering standards.

Table 3.4 organizes our evaluation framework into nine distinct tasks spanning both requirement types. Functional tasks (F1-F4) assess objective code correctness across four established benchmarks, progressing from basic generation to complex reasoning scenarios. Non-functional tasks (NF1-NF5) evaluate qualitative aspects through the comprehensive CODAL benchmark, covering the standard software engineering best practices.

Functional Requirements in Coding Tasks

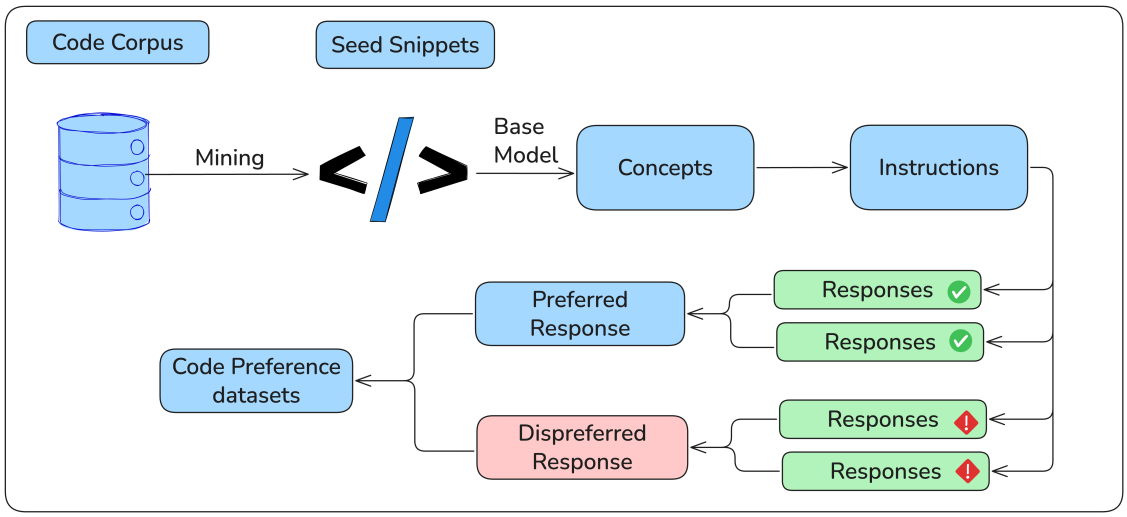


Figure 3.6: Data Generation Workflow

Functional requirements specify what the code must accomplish to be executable and produce correct outputs. This includes generating syntactically correct, compilable code in target programming languages that follows established standards, incorporates proper error handling (Figure 3.7), implements security practices, and includes performance optimization.

```python
## With proper error handling
def remove_duplicates(lst):
    if not isinstance(lst, list):
        raise TypeError("Input must be a list")
    if not lst:
        return []
    return list(dict.fromkeys(lst))

# Without error handling
def remove_duplicates(lst):
    return list(dict.fromkeys(lst))  # Crashes on None input
```

Figure 3.7: Error handling

Non-functional Requirements

Non-functional Requirements primarily include instruction following, code explanation, code complexity and efficiency, code readability (Figure 3.8) , and coding style as defined in [19]. These requirements ensure the generated code meets broader software engineering standards

and practices. These five dimensions are derived from established software engineering practices rather than direct developer elicitation, representing benchmark-defined proxies for code quality. This categorization allows for a more comprehensive evaluation of code LLM alignment across both correctness aspects (functional) and quality aspects (non-functional).

Our methodology integrates prior work on LLM alignment and code generation, notably SelfCodeAlign [11] and BoNBoN [51]. We utilize SelfCodeAlign to generate fully transparent training preference datasets for Direct Preference Optimization (DPO) and BoNBoN LLM alignment techniques, as shown in Figure 3.6.

3.3.2 Creation of Preference Datasets

SelfCodeAlign proposes a pipeline to generate fully transparent instruction tuning datasets for supervised fine-tuning. In this work, we modify the pipeline to generate preference datasets for LLM alignment. The process begins with seed functions generated from the Stack V1 source code dataset [43]. Next, using a selected base model, coding concepts are extracted. Using these coding concepts, instructions are generated to reconstruct the original code snippet. The same base model then generates multiple responses for each instruction.

These responses are executed and filtered to include only passing responses in the instruction tuning dataset. We modify this pipeline by randomly selecting one passing response as the chosen response and one failing non-empty response as the rejected response. This approach enables us to build preference datasets for each model. Note that we generate preference datasets separately for each model because LLMs learn more efficiently when training data follows the same distribution as the base model's data distribution [11].

```python
## Readable implementation
def longest_common_prefix(strings):
    if not strings:
        return ""

    shortest_string = min(strings, key=len)

    for index, char in enumerate(
        shortest_string):
        for string in strings:
            if string[index] != char:
                return shortest_string[:
                    index]

    return shortest_string

# Unreadable implementation
def lcp(s):
    if not s:return ""
    m=min(s,key=len)
    for i,c in enumerate(m):
        for st in s:
            if st[i]!=c:return m[:i]
    return m
```

Figure 3.8: Code Readability

For non-functional requirements, we generate separate preference datasets for each dimension using dimension-specific prompts. We applied similar modifications across all five evaluation dimensions, ensuring balanced representation (e.g., a 60k non-functional dataset contains 12k samples per dimen-

sion). Our results validate this approach: code-specialized models achieved consistent improvements across all five dimensions (O3), and different dimensions showed distinct response patterns (O6), suggesting the preference pairs capture meaningful quality differences.
Using the above approach, we generated datasets for each model. Figure 3.5 shows the models and the number of preference instances for each model.

3.3.3 Model Selection

Our study evaluates five base open source models along with their corresponding instruction-tuned versions, for a total of 10 models. The selected models include Meta-Llama-3-8B [32], Qwen2.5-Coder-7B[29], CodeLlama-7b [30], deepseek-coder-1.3b [5], and deepseek-coder-7b [5]. These are paired with their respective instruction-tuned [33] versions: Meta-Llama-3-8B-Instruct, Qwen2.5-Coder-7B-Instruct, CodeLlama-7b-Instruct-hf, deepseek-coder-1.3b-instruct, and deepseek-coder-7b-instruct-v1.5.
Our model selection prioritizes diversity among selected models while maintaining a small resource footprint. The selection consists of different sizes ranging from 1.3B to 8B parameters, providing insight into how model scale affects alignment effectiveness. We include both general-purpose models (Meta-Llama-3-8B) and code-specialized models (Qwen2.5-Coder, CodeLlama, deepseek-coder), enabling us to investigate whether domain specialization influences alignment outcomes. While other open source variants exist, our selection focuses on models that are: (1) actively maintained with stable releases, (2) widely adopted as benchmarks, and (3) representative of distinct architectures. This diversity allows us to investigate how different architectures affect the effectiveness of alignment pathways while ensuring computational feasibility through our focus on smaller models.

3.3.4 Evaluation benchmarks

Evaluation on Functional benchmarks

HumanEval+ / MBPP+ are functional-level coding benchmarks. HumanEval+ extends the original HumanEval [15] with 164 instances and 80x more test cases, while MBPP+ enhances MBPP [17] with 378 instances and 35x more testing coverage. We use evalplus [36] for standardized evaluation with Pass@1 score.
EvalPerf assesses LLM-generated code efficiency alongside HumanEval+ and MBPP+. This 118-instance benchmark [36] measures the percentage of solutions meeting reference efficiency.
Evoeval provides contamination-free assessment through systematically modified problems. The benchmark contains 828 problems across seven categories, evaluated using Pass@1 score [18].

Evaluation on Non-functional benchmarks

CODAL [19] is a non-functional requirements benchmark with 500 problems across five dimensions: instruction-following, code explanation, complexity/efficiency, readability, and coding style. Using LLM-as-judge methodology, it provides 0-10 ratings per dimension and aggregate scores.

3.3.5 Evaluation Metrics

All of our results in the section 4.5 are reported as relative improvement as defined below:

$$\Delta_{rel,i} = \frac{P_{final,i} - P_{baseline,i}}{P_{baseline,i}} \times 100\% \tag{3.1}$$

where $P_{final,i}, P_{baseline,i}$ are aligned/baseline Pass@1 (functional) or CODAL scores (non-functional).

Pass@1

Percentage of problems solved correctly on first attempt (HumanEval/+, MBPP+, EvoEval).

CODAL Score

LLM judge rates responses 1-10 across 5 code quality dimensions (instruction-following, explanation, efficiency, readability, style). Individual dimension score S_p averages ratings across 100 problems; overall score averages across dimensions. $r_{p,i} \in \{1, \ldots, 10\}$ = LLM judge rating for problem i in dimension p. $p \in \{1, \ldots, 5\}$ = code quality dimensions.

$$S_p = \frac{1}{100} \sum_{i=1}^{100} r_{p,i}, \quad \text{CODAL Score} = \frac{1}{5} \sum_{p=1}^{5} S_p \tag{3.2}$$

3.3.6 Threats to Validity

Limited model coverage (five base models, 1.3B-8B parameters) may not represent larger models exceeding 70B parameters, which could exhibit emergent properties [79] and could alter alignment effectiveness [80]. Utilzing those models, however, will require significant GPU resources. Nevertheless, our findings and study setups could guide companies with bigger models working on code LLMs. Our evaluation emphasizes general-purpose code generation tasks, potentially missing specialized programming categories like class-level generation or bug fixes that may exhibit different quality requirements. However, alignment objectives for such tasks are also not standardized, i.e., we may not need alignment for such tasks anyway. Construct validity concerns arise from our operational definition of alignment through functional and non-functional requirements, which may not encompass all dimensions practitioners consider essential, such as robustness to adversarial conditions or ethical compliance.

Additionally, the CODAL benchmark dimensions represent software engineering practices rather than directly validated developer priorities; while these dimensions align with recognized code quality standards, they serve as proxies for code quality rather than developer preferences. All preference pairs in our study are model-specific and self-generated, following SelfCodeAlign's finding that models learn more efficiently from matching distributions [11]. While this maximizes within-model

alignment effectiveness, it may encode model-specific stylistic biases rather than broadly generalizable quality patterns. Cross-model preference transfer remains an important direction for future work.

3.4 Results

3.4.1 Performance Comparison between Pretrained-to-Aligned (RQ1)

We analyze how alignment techniques (DPO, BoNBoN) affect pretrained code LLMs across functional (RQ 1.1) and non-functional (RQ 1.2) coding tasks as detailed in Table 3.4.

Functional Coding Tasks in Pretrained Models (RQ 1.1)

Preference-based alignment produces highly variable functional improvements across coding tasks F1-F4, with some models achieving higher relative improvement despite starting from significantly lower absolute performance baseline scores compared to instruction-tuned variants. Results are shown in Table 3.1 for F1-F3 tasks and Table 3.3 for F4 comprehensive evaluation, and all results are reported in Pass@1 scores below.

Success Cases Llama3-8b demonstrates the strongest improvements across all functional coding tasks, with basic code generation (F1) improving 42% (0.372→0.530) with DPO and 47% on HumanEval+ with DPO/BoNBoN, algorithmic problem solving (F2) advancing 9.5% (0.630→0.690) with DPO/BoNBoN, and performance optimization (F3) achieving 82% improvement (6.5%→11.8%) using BoNBoN. Comprehensive evaluation (F4) confirms these broad improvements with difficult tasks improving 94% with BoNBoN, creative tasks advancing 48% with DPO, and ToolUse capabilities increasing 35% with BoNBoN, indicating that alignment successfully enhances functional capabilities when models have sufficient starting point scores (≥ 0.1).

Observation 1: Pretrained models achieve 5.9x larger relative improvements (28.8% vs 4.9% average) despite lower baseline performance compared to instruction-tuned versions, and are 1.3x more likely to benefit from alignment (67.1% vs 51.4% success rates), demonstrating higher plasticity across all functional task categories F1-F4.

Consistent Improvement Cases Qwen2.5-7b shows steady functional improvements across alignment methods with basic generation (F1) improving 4.2% (0.720→0.750) with BoNBoN and 6.3% on HumanEval+, mixed algorithmic results with MBPP declining 1.7% (0.759→0.746), and performance optimization (F3) increasing 0.4 percentage points with DPO. Comprehensive evaluation (F4) maintains consistent patterns with difficult tasks, improving 5.1% and creative tasks advancing 18.4% with DPO, suggesting that higher-baseline models achieve consistent functional improvements without degradation.

Observation 2: Baseline performance determines alignment viability across all functional coding tasks F1-F4: models with extremely low functional baseline scores show fundamental limitations preventing effective alignment, while models above the minimum threshold on average ($\geq$ Pass@1 score) achieve substantial improvements.

Non-Functional Coding Tasks in Pretrained Models (RQ 1.2)

Our analysis demonstrates that non-functional alignment effectiveness varies significantly by model type across all task dimensions NF1-NF5, with code-specialized models achieving substantial improvements (avg 36.2%) while general-purpose models show minimal gains. Results are shown in Table 3.2 and all results are reported as CODAL scores (0-10 scale) below.

Success Cases: Code-Specialized Models. Qwen2.5-Coder-7B demonstrates significant improvements with average CODAL scores improving 15.7% (1.27→1.47) with BoNBoN, including instruction following (NF1) advancing 31.5%, complexity/efficiency (NF3) enhancing 21.2%, and coding style (NF4) improving 22.3%. deepseek-coder-7b exhibits similar positive improvements with average CODAL scores improving 27.8%, while CodeLlama-7b achieves the most substantial relative improvement with average CODAL scores increasing 75.0%, including exceptional improvements in coding style (NF4) advancing 82.0% and code readability (NF2) improving 82.0%.

Observation 3: Code-specialized models significantly outperform general-purpose models in the Pretrained-to-aligned pathway across all non-functional coding tasks NF1-NF5, achieving 16-75% relative improvement, while general-purpose models show minimal meaningful improvement.

Limited Success: General-Purpose Models Meta-Llama-3-8B shows modest improvement primarily through SFT with average CODAL scores advancing 66% (0.03→0.05) relative but only 0.02 points. The extremely low baseline scores (0.01-0.09 range) suggest fundamental limitations in code-specific non-functional properties for general-purpose models, with preference-based methods maintaining baseline performance without significant improvement.

Observation 4: BoNBoN generally outperforms DPO for Pretrained-to-aligned non-functional improvement across multiple models and non-functional task categories NF1-NF5. winning 76.7% of the time vs DPO's 10.0% of the time, with 53.3% average improvement when BoNBoN wins vs 22.1% when DPO wins.

Small Model Limitations deepseek-coder-1.3b demonstrates limited but consistent improvement with average CODAL scores advancing 26.3% (0.19→0.24) with BoNBoN. While relative improvements are substantial, absolute performance remains constrained by model size.

Observation 5: Alignment responsiveness varies by model size within the same model family. Larger variants show measurable improvements while smaller models exhibit limited improvements (6.4x for DeepSeek 7B vs 1.3B), indicating scale-dependent alignment capacity.

Observation 6: Non-functional tasks show distinct response patterns - coding style (NF4) consistently improves substantially (avg 108%), code complexity (NF3) and explanation (NF5) vary widely, while instruction following (NF1) shows strong but inconsistent improvements.

Table 3.1: Functional Benchmarks Performance (Pass @1): Pre-trained vs Fine-Tuned Comparison, Pre = Pre-trained models, Fine = Fine-tuned models, HE = HumanEval , HE+ = HumanEval+

	F1 (HE)		**F1 (HE+)**		**F2 (MBPP)**		**F2 (MBPP+)**		**F3 (EvalPerf)**	
Model	**Pre**	**Fine**	**Pre**	**Fine**	**Pre**	**Fine**	**Pre**	**Fine**	**Pre**	**Fine**
Qwen2.5-7B										
Base	0.720	0.854	0.671	**0.811**	**0.759**	0.799	**0.661**	0.669	63.4%	71.1%
SFT	0.726	**0.860**	0.689	0.805	0.749	0.791	0.646	0.675	63.5%	72.6%
DPO	0.738	0.854	0.701	**0.811**	0.733	**0.810**	0.635	**0.683**	**63.8%**	73.3%
BoNBoN	**0.750**	0.854	**0.713**	**0.811**	0.746	0.804	0.643	0.677	63.3%	**73.5%**
Llama3-8B										
Base	0.372	**0.610**	0.323	**0.549**	0.630	**0.646**	0.526	**0.542**	6.5%	**57.4%**
SFT	0.470	0.579	0.421	0.524	0.664	0.601	0.558	0.508	9.7%	53.6%
DPO	**0.530**	0.561	**0.476**	0.506	**0.690**	0.630	0.545	0.537	11.5%	56.4%
BoNBoN	0.518	0.543	**0.476**	0.482	**0.690**	0.614	**0.574**	0.524	**11.8%**	57.3%
CodeLlama-7B										
Base	**0.030**	0.390	**0.018**	0.317	0.556	**0.548**	0.439	**0.458**	0.8%	43.7%
SFT	0.012	**0.409**	0.012	**0.335**	**0.608**	0.532	**0.479**	0.439	0.9%	45.3%
DPO	0.018	0.372	0.012	0.317	**0.608**	0.537	0.481	0.442	**1.0%**	**47.3%**
BoNBoN	0.012	0.360	0.006	0.299	0.590	0.540	0.466	0.444	0.8%	46.8%
DeepSeek-1.3B										
Base	0.335	**0.665**	0.280	**0.628**	**0.579**	0.622	**0.492**	0.521	3.8%	56.5%
SFT	0.341	0.622	0.299	0.585	0.577	0.622	0.489	**0.529**	**4.1%**	57.0%
DPO	0.341	**0.665**	0.299	**0.628**	**0.579**	**0.630**	0.489	0.524	3.9%	57.0%
BoNBoN	**0.366**	0.640	**0.317**	0.598	**0.579**	0.619	0.489	0.521	4.0%	**57.3%**
DeepSeek-7B										
Base	**0.463**	0.720	**0.396**	0.640	0.712	**0.757**	0.606	**0.664**	6.4%	72.7%
SFT	0.427	**0.738**	0.366	**0.659**	0.717	0.741	0.603	0.648	6.5%	72.8%
DPO	0.439	0.732	0.378	**0.659**	0.725	0.749	0.611	0.653	**6.7%**	**73.6%**
BoNBoN	0.415	0.695	0.354	0.640	**0.730**	0.749	**0.614**	0.651	6.3%	73.3%

Table 3.2: CODAL benchmark results (rating out of 10): Pre-trained vs Fine-Tuned Comparison. Pre = Pre-trained models, Fine = Fine-tuned models. IF = Instruction Following, CR = Code Readability, CCE = Code Complexity and Efficiency, CS = Coding Style, CE = Code Explanation, Avg = Average

Model	**NF1 (IF)**		**NF2 (CR)**		**NF3 (CCE)**		**NF4 (CS)**		**NF5 (CE)**		**Avg**	
	Pre	**Fine**	**Pre**	**Fine**	**Pre**	**Fine**	**Pre**	**Fine**	**Pre**	**Fine**	**Pre**	**Fine**
meta-llama/Meta-Llama-3-8B												
Base	0.01	3.72	0.04	6.03	0.02	5.21	0.01	5.93	0.09	6.19	0.03	5.42
SFT	**0.02**	4.06	**0.06**	4.52	0.02	5.70	**0.08**	5.42	**0.07**	6.05	**0.05**	5.15
DPO	0.01	5.39	0.04	**6.45**	0.02	6.85	0.04	**6.45**	0.02	**7.53**	0.03	**6.53**
BoNBoN	0.01	**5.43**	0.04	6.42	0.02	**6.89**	0.05	6.32	0.03	7.24	0.03	6.46
Qwen/Qwen2.5-Coder-7B												
Base	1.11	6.25	1.31	**8.35**	1.32	7.74	1.48	**7.29**	1.14	8.32	1.27	**7.59**
SFT	1.20	**6.33**	**1.41**	8.27	1.45	7.66	1.56	7.15	1.09	**8.48**	1.34	7.58
DPO	**1.51**	6.21	1.22	8.22	1.43	7.66	1.72	7.17	**1.22**	**8.48**	**1.42**	7.58
BoNBoN	1.46	6.01	1.32	8.13	**1.60**	**7.75**	**1.81**	7.10	1.16	8.41	1.47	7.48
meta-llama/CodeLlama-7b												
Base	0.51	**0.76**	0.50	**0.64**	0.45	**0.40**	0.50	**0.69**	0.46	**0.53**	0.48	**0.60**
SFT	0.82	0.53	0.85	0.47	0.71	0.24	0.84	0.46	0.65	0.32	0.77	0.40
DPO	0.80	0.55	0.86	0.50	0.68	0.22	0.79	0.55	0.64	0.36	0.75	0.44
BoNBoN	**0.88**	0.50	**0.91**	0.46	**0.78**	0.18	**0.91**	0.41	**0.74**	0.32	**0.84**	0.37
deepseek-ai/deepseek-coder-1.3b												
Base	0.11	**3.16**	0.07	**4.48**	0.31	3.27	0.32	3.09	0.14	2.77	0.19	**3.35**
SFT	0.15	3.10	0.09	4.04	0.33	**3.35**	**0.33**	3.10	0.16	3.09	0.21	3.34
DPO	0.10	2.99	0.06	4.15	0.30	3.30	0.29	**3.27**	0.16	2.87	0.18	3.32
BoNBoN	**0.18**	3.08	**0.13**	4.14	**0.37**	3.09	0.32	3.24	**0.19**	**3.11**	**0.24**	3.33
deepseek-ai/deepseek-coder-7b												
Base	1.26	**5.38**	1.16	7.04	1.10	6.06	0.96	6.03	1.20	6.45	1.15	6.19
SFT	1.31	**5.38**	1.12	**7.14**	1.34	**6.23**	1.20	5.98	**1.48**	6.57	1.29	**6.26**
DPO	1.44	5.10	1.37	7.12	1.30	6.18	1.21	5.92	**1.48**	**6.68**	1.36	6.20
BoNBoN	**1.55**	5.27	**1.51**	**7.14**	**1.50**	6.26	1.34	**5.78**	1.47	6.65	**1.47**	6.22

3.4.2 Performance Comparison between Finetuned-to-Aligned (RQ2)

We examine how preference-based alignment techniques affect already fine-tuned LLMs across both functional (RQ 2.1) and non-functional (RQ 2.2) coding tasks as separate research questions. Our empirical investigation demonstrates that fine-tuned LLM alignment effectiveness depends on both the target objective and base model properties. While alignment for non-functional requirements shows consistent improvements, functional alignment shows highly variable outcomes ranging from significant improvements to catastrophic degradation.

Functional Coding Tasks in Finetuned Models (RQ 2.1)

Functional alignment performance varies across models, with outcomes ranging from significant improvement to catastrophic failures, as shown in Table 3.1: EvalPlus, Table 3.3: EvoEval. This suggests that the base model architecture and pretraining step, as well as data distribution, signif-

Table 3.3: EvoEval Benchmark Results (Pass @1): Pre-trained (P) vs Fine-Tuned (F) Comparison. Diff = Difficult, Crea = Creative, Sub = Subtle, Comb = Combine, Tool = ToolUse, Verb+ = Verbose+, Conc+ = Concise+.

Model	Diff		Crea		Sub		Comb		Tool		Verb+		Conc+	
	P	**F**	**P**	**F**	**P**	**F**	**P**	**F**	**P**	**F**	**P**	**F**	**P**	**F**
Qwen2.5-Coder-7B														
Base	.390	**.510**	.380	.430	**.710**	**.840**	.250	.370	.430	**.650**	.622	.768	.713	.768
SFT	.400	.440	.420	.430	.700	.770	**.310**	.370	.450	.620	.689	.793	.713	.780
DPO	**.410**	.490	**.450**	.420	.680	.780	.290	.360	**.470**	**.650**	**.701**	**.805**	.707	**.793**
BoNBoN	.380	.450	.380	**.440**	.690	.780	**.310**	**.380**	.450	.630	.677	.793	**.738**	.787
Meta-Llama-3-8B														
Base	.170	.320	.250	**.390**	.370	**.590**	.050	**.150**	.370	**.560**	.354	**.567**	.335	**.530**
SFT	.220	.290	.310	.370	**.470**	.530	.100	**.150**	.450	.460	.476	.549	.415	.500
DPO	.210	.320	**.370**	.310	**.470**	.510	.110	.130	.460	.490	.445	.530	**.445**	.470
BoNBoN	**.330**	**.330**	.310	.310	.510	.510	**.140**	.140	**.500**	.500	**.478**	.549	.451	.500
CodeLlama-7b														
Base	.000	.100	.010	.170	.020	.350	**.020**	**.110**	**.040**	**.410**	**.134**	.335	**.006**	**.360**
SFT	.010	.110	.020	.170	.000	**.370**	.000	.060	.020	.370	.091	**.341**	.000	.354
DPO	**.020**	**.140**	**.030**	**.210**	.000	.340	.010	.070	.020	.360	.122	.323	.000	.311
BoNBoN	.000	**.140**	.020	.200	.000	.350	.000	.040	.020	.350	.079	.299	.000	.317
DeepSeek-Coder-1.3B														
Base	.060	.180	.160	**.260**	.330	**.530**	.000	.080	**.400**	.430	.317	.567	.293	.591
SFT	.080	.150	.170	.240	**.360**	.520	.000	**.100**	.390	**.450**	.311	**.604**	.311	.585
DPO	**.090**	**.190**	**.190**	.250	.330	.520	.010	**.100**	**.400**	.440	.293	.598	.299	**.598**
BoNBoN	.080	.170	**.190**	.220	**.360**	**.530**	**.020**	.090	.390	.420	**.329**	.573	**.323**	.585
DeepSeek-Coder-7B														
Base	.190	**.400**	.310	**.400**	.440	.640	.080	**.320**	.480	**.650**	**.494**	.634	**.409**	**.677**
SFT	.150	.380	**.330**	.380	**.450**	**.690**	.090	.220	**.530**	.640	.482	.659	.384	.622
DPO	**.210**	.380	.300	**.400**	.420	.660	**.110**	.210	.510	.640	.482	**.671**	.402	.659
BoNBoN	.170	.340	.280	.370	.410	**.690**	.080	.260	.520	.620	.488	.665	.402	.646

icantly impact the alignment effectiveness.

Success Cases deepseek-7b achieves consistent improvements with efficiency evaluation (F3) increasing from 72.7% to 73.6%, basic code generation (F1) improving +1.7%, and minimal algorithmic degradation (0.757→0.749). On comprehensive evaluation (F4), deepseek-7b maintains stable performance on Difficult tasks with minor degradation -5.0% while showing clear Verbose+ improvements +5.8%. Qwen2.5-7b shows significant improvements with BoNBoN achieving the highest efficiency score (73.5%) across all models, representing a 2.4 percentage point improvement from base (71.1%), and demonstrates comparable F4 stability with +4.8% improvements with DPO.

Observation 7: Functional alignment in finetuned models shows highly variable outcomes with 47.4% degradation cases, 36.8% flat performance, and only 15.8% net improvements, indicating strong model dependency and diminishing returns in alignment effectiveness.

Failure Cases Llama3-8b shows performance degradation after SFT with efficiency evaluation (F3) dropping from 57.4% to 53.6%, basic generation (F1) decreasing -5.1%, and algorithmic performance falling -7.0%. Subsequent DPO and BoNBoN fail to recover performance, remaining around 56.4-57.3%. This degradation extends to comprehensive assessment (F4) with decreased performance across most dimensions, particularly Difficult tasks -9.4% and Creative tasks -5.1%.

Observation 8: SFT performance degradation is a significant risk indicator for alignment failure with only a 25.0% full recovery rate. models degrading during supervised fine-tuning frequently struggle to achieve full recovery with 4 times more higher failure risk.

Observation 9: Alignment training creates performance trade-offs within individual models on F4 - CodeLlama-7b-dpo improves on Difficult (0.100→0.140) and Creative (0.170→0.210) tasks while declining on Verbose (0.427→0.402) and Concise (0.409→0.378) tasks.

Observation 10: Models that maintain consistent performance across different coding tasks during alignment training (i.e, Qwen2.5-7b) are more reliable than models showing large performance changes (i.e, CodeLlama-7b), indicating that steady performing models are less likely to fail on new tasks.

Non-Functional Coding Tasks in Finetuned Models (RQ 2.2)

Non-functional alignment demonstrates consistent effectiveness across different already fine-tuned models compared to functional alignment, as shown in Table 3.2: CODAL benchmark. All results are reported as CODAL scores (0-10 scale).

Under Aligned Models Llama3-8B shows significant improvements across all non-functional dimensions with average CODAL scores increasing from 5.42 to 6.53 with DPO (20% relative improvement). Individual dimensions show instruction following (NF1) improving +45% (3.72→5.39), code explanation quality (NF5) increasing +22%, and complexity/efficiency (NF3) improving +31%, with DPO achieving the highest overall score (6.53) compared to BoNBoN (6.46).

Observation 11: Non-functional alignment (NF1-NF5) demonstrates more consistent effectiveness across finetuned models, unlike the variable functional alignment results.

Saturated Improvement Qwen2.5-7b shows performance saturation with a base CODAL score of 7.59, representing near-optimal performance. Alignment tuning provides only minimal improvement, suggesting the model is already well-optimized for non-functional requirements, with slight variations indicating negligible benefits for already well-aligned models.

Observation 12: Under-aligned models show significant non-functional improvements (Llama3-8B: 20%), while well-aligned models exhibit performance saturation (Qwen2.5-Coder-7B, 1.1%).

Failure cases CodeLlama-7b shows anomalous results where alignment severely degrades non-functional performance across all tasks NF1-NF5, with base score dropping from 0.60 to 0.40-0.44. Specific degradations include instruction following -34%, code readability -28%, complexity/efficiency -55%, coding style -41%, and code explanation -40%, suggesting fundamental incompatibilities between model architecture and alignment techniques.

Observation 13: Certain base models show fundamental incompatibilities with alignment methods, resulting in catastrophic performance degradation (CodeLlama-7b: -38.3%).

3.4.3 Comparative Analysis (RQ3)

We compare these approaches across models and coding tasks to quantify the relative improvement each pathway provides. We answer RQ3 in two parts, Pathway Trade-offs in Alignment Outcomes (RQ 3.1) and Starting Point Effects on Alignment Techniques (RQ 3.2), enabling us to identify trade-offs and draw conclusions about optimal alignment strategies.

Pathway Trade-offs in Alignment Outcomes (RQ 3.1)

Pretrained-to-Aligned Shows Higher Relative Improvement. Pretrained-to-aligned models demonstrate significantly larger improvement compared to finetuned-to-aligned, particularly for non-functional coding tasks (NF1-NF5). Pretrained models achieved remarkable non-functional improvements: CodeLlama-7b from 0.48 to 0.84 (+75%), deepseek-coder-7b from 1.15 to 1.47 (+27.8%), and Qwen2.5-Coder-7B from 1.27 to 1.47 (+15.7%). In contrast, finetuned-to-aligned models showed constrained improvements, with some experiencing degradation (CodeLlama-7b-Instruct). For functional tasks (F1-F4), pretrained models again showed larger relative improvements despite lower absolute starting points, with Llama3-8b achieving 42% improvement in basic generation while its instruction-tuned counterpart experienced degradation (-5%).

The stability-plasticity dilemma. Our findings align with the stability-plasticity dilemma [81] observed in LLM continual learning research, showing fundamental trade-offs between learning plasticity (fast adaptation) and memory stability (knowledge preservation). Base pre-trained models show higher plasticity, enabling significant capability improvements but risk catastrophic forgetting, while instruction-tuned models demonstrate higher stability but reduced plasticity, constraining achievable improvements. Performance saturation in well-aligned models like Qwen2.5-Coder-7B-Instruct (7.59→7.58) clearly demonstrates this trade-off.

Observation 14: Alignment pathways exhibit a stability-plasticity: base models show high plasticity/low stability, enabling significant improvements (avg +42.3%), while instruction-tuned models show high stability/low plasticity with limited improvements (avg +42.3%).

Observation 15: Non-functional coding tasks NF1-NF5 demonstrate higher alignment responsiveness (60% success rate) across both pathways compared to functional tasks F1-F4, which show variable responsiveness (20% success rate).

Starting Point Effects on Alignment Techniques (RQ 3.2)

Our analysis reveals that optimal alignment technique selection depends primarily on the specific learning objective rather than following universal heuristics based on starting points. The effectiveness patterns show complex model-dependent patterns.

Non-Functional Alignment Shows Technique-Dependent Patterns: The largest single improvement occurred with DPO on instruction-tuned Meta-Llama-3-8B-Instruct (+1.11 across NF1-NF5), showing DPO's effectiveness for finetuned-to-aligned non-functional tasks. However, base models show mixed results: BoNBoN achieved remarkable improvements on Qwen2.5-Coder-7B (+0.20 vs +0.15) and deepseek-coder-7b (+0.32 vs +0.21), while both techniques failed completely on Meta-Llama-3-8B base model (0.00 improvement).

Observation 16: Alignment effectiveness varies more by task type than complexity - functional tasks show variable responsiveness (F1: 60% success), while non-functional tasks demonstrate consistent responsiveness regardless of complexity (NF1-NF5: 80-100% success rates)

Functional Alignment Reveals Model-Specific Preferences: Functional improvements demonstrate strong model dependency. Meta-Llama-3-8B consistently performs better on DPO across both pathways (base: +0.158 vs +0.146 on F1; instruction-tuned: -0.049 vs -0.067), while smaller models like deepseek1.3b show BoNBoN advantages in base settings (+0.031 vs +0.006 in F1) but degradation in the instruction-tuned pathway (-0.025 vs 0.000).

Observation 17: Alignment technique preferences show model family-specific patterns that vary by pathway: Llama favors DPO for a subset of tasks while deepseek prefers BonBon primarily in the pretrained-to-aligned pathway, but both families demonstrate technique preference across different starting points and objectives.

Observation 18: Optimal alignment technique selection depends on both learning objective and model family (non-functional: Llama3→DPO, deepseek→SFT, Qwen2.5→base) rather than a universal pattern: identical objectives require different techniques across model families, indicating complex model-dependent alignment effectiveness.

Table 3.4: Recommendations and Supporting Observations, O = Observation

ID	Recommendation	O
Category 1. Pathway Selection		
R1	Consider choosing pretrained-to-aligned pathways when maximizing improvement magnitude is prioritized; finetuned-to-aligned pathways may be preferable when stable, high absolute performance is required	1, 14
R2	We recommend focusing on non-functional alignment objectives first, as they tend to achieve higher success rates and more predictable results	6, 11, 15
Category 2. Model Selection		
R3	It is advisable to prioritize code-specialized models over general models and consider selecting larger variants ($\geq$7B parameters) within model families	3, 5, 20
R4	We suggest establishing minimum performance thresholds and assessing baseline alignment quality before attempting alignment, with efforts concentrated on under-aligned models	2, 12
Category 3. Technique Selection		
R5	Consider applying model family-specific technique preferences (Meta-Llama→DPO, deepseek→BoNBoN) and conducting empirical validation for each model-objective combination	4, 17, 18
R6	Using diverse benchmarks during evaluation is recommended, as alignment effectiveness varies by task complexity	16
Category 4. Risk Management		
R7	Early stopping based on SFT-stage performance degradation should be considered, as it tends to strongly predict final alignment failure	8
R8	We recommend monitoring multi-dimensional performance during alignment to detect capability trade-offs and selecting models with stable performance	9, 10
R9	Pre-screening models for compatibility is advisable, along with implementing stronger safeguards for instruction-tuned models to prevent failures	7, 13, 19

Alignment Risks Vary by Starting Point and Technique. Both DPO and BoNBoN frequently cause functional degradation when applied to instruction-tuned models, including CodeLlama-7b-Instruct dropping from 0.390 to 0.360 (BoNBoN) and 0.372 (DPO) in F1, and non-functional degradation from 0.60 to 0.37 (BoNBoN) and 0.44 (DPO). Pretrained models demonstrate more consistent improvement patterns with lower catastrophic failure rates, showing minimal degradation even when improvements are small.

Observation 19: Alignment degradation risk varies significantly by starting point and technique: instruction-tuned models show higher catastrophic failure rates (20%), while pretrained models show more predictable degradation risks (0%).

Observation 20: Model architecture and size correlate more strongly with alignment success than starting point selection.

3.5 Recommendations

In Table 3.4, we present nine recommendations based on our empirical observations (O) in Section 3.4. We organized the recommendations into four categories and discuss those below.

Pathway Selection (R1, R2). We recommend pretrained-to-aligned pathways when maximizing improvement is the primary goal, as these tend to achieve significant relative improvement despite lower baseline values. Conversely, it is also beneficial to select finetuned-to-aligned pathways when stable, high absolute performance is required (O1, O14). Regardless of pathway choice, We suggest prioritizing non-functional alignment objectives first, as they demonstrate higher success rates and more predictable outcomes than functional requirements (O6, O11, O15).

Model Selection (R3, R4). In terms of model selection, We recommend prioritizing code-specialized models over general-purpose alternatives and considering larger variants within model families ($\geq$ 7B parameters)(O3, 5, 20). Building on this foundation, it is also advisable to establish a minimum performance threshold and assess baseline alignment quality before initiating alignment processes. This assessment allows practitioners to focus efforts on under-aligned models where improvements are most likely to succeed (O2, 12).

Technique Selection (R5, R6). Our findings suggest that Meta-Llama models consistently respond better to DPO, while deepseek models demonstrate better performance with BoNBoN (O4, 17, 18). However, these patterns should serve as starting points rather than absolute rules. We recommend conducting empirical validation for each specific model-objective combination to confirm optimal technique selection. Additionally, consider using diverse evaluation benchmarks throughout alignment, as effectiveness varies significantly across different tasks (O16).

Risk Management (R7, R8, R9). Throughout the alignment process, proactive risk management helps prevent costly failures and performance degradations. We recommend considering early stopping mechanisms based on SFT-stage performance degradation, which strongly predicts final alignment failure (O8). It is also beneficial to monitor multi-dimensional performance across evaluation benchmarks to detect capability trade-offs and select models demonstrating consistent improvement (O9, 10). Finally, we suggest pre-screening model architectures for compatibility and implementing stronger safeguards for instruction-tuned models, as certain families may experience severe performance degradation (O7, 13, 19).

3.6 Discussions

3.6.1 Practitioner Validation

To empirically validate the practical significance of the trade-offs identified in our study, we conducted a survey with 30 ML practitioners and software engineers from diverse organizations and several universities. Table 3.5 shows the survey questions. The survey was conducted under an ethics application at the university. We adopted a snowball sampling approach to collect the study participants [82]. First, we selected industry participants from personal contacts in companies that focus on this area (e.g., H2O). Those participants then invited other relevant contacts. The survey was open for around one month (Jan 5, 2026 to Jan 31, 2026), due to it being held close to the holiday season. 53.3% of the participants had 3-5 years of relevant experience (e.g., ML engineering). 46.7% already performed fine-tuning and alignment on LLMs (Q1-Q3). To ensure that the

Table 3.5: Practitioner Survey Questions for Validation Study

ID	Question	Response Options
Demographics		
Q1	Primary role	ML Engineer/Researcher, Software Engineer, Research Scientist, Other
Q2	Years of experience	<1 year, 1-2 years, 3-5 years, >5 years
Q3	Alignment training experience	No experience, Done once/twice, Familiar but not done, Done multiple times
PTA vs. FTA Trade-offs		
Q4	Typical starting point for LLM adaptation	Base pretrained (PTA), Instruction-tuned (FTA), Evaluate both, Not applicable
Q5	Importance of pathway choice	Very important, Somewhat important, Not very important, Not applicable
Q6	Factors influencing choice (multi-select)	Task requirements, Compute resources, Time constraints, Expected improvement, Degradation risk, Prior experience
Relative vs. Absolute Performance		
Q7	Metric prioritization	Relative improvement, Absolute performance, Both (prioritize relative/absolute), N/A
Q8	Scenario preference	Option A: 40% relative, 80% overall vs. Option B: 5% relative, 85% overall

participants understood the survey, we produced an online tutorial that explained LLM code alignment techniques. Each participant read the tutorial before completing the survey. The anonymized survey responses are included in our replication package.

Significance of PTA vs. FTA Trade-offs (Q4 - Q6)

We asked about the importance of choosing between pretrained-to-aligned (PTA) and finetuned-to-aligned (FTA) pathways (Q5). **86.7%** of respondents rated this choice as "Very important" (40.0%) or "Somewhat important" (46.7%). Furthermore, 33.3% of practitioners reported that they actively evaluate both options depending on the use case (Q4), confirming that this is a decision point rather than a fixed practice. When asked about factors influencing their pathway selection (Q6), practitioners identified target task requirements (80%), compute resources (73.3%), time constraints (53.3%), and expected improvement magnitude (46.7%) as key considerations; aligning with our recommendations in Section 3.5.

Relative vs. Absolute Performance Metrics (Q7, Q8)

To validate whether practitioners value relative improvements alongside absolute performance, we assessed metric prioritization (Q7). **50.0%** prioritize relative improvement while **40%** prioritize absolute performance, demonstrating that both metrics are relevant in practice. We further presented a concrete scenario (Q8) reflecting the PTA vs. FTA trade-off: practitioners were asked which alignment outcome they would prefer for a production deployment: (A) 40% relative improvement in non-functional properties (security, compliance, readability) reaching 80% overall, representing a

PTA-like outcome, or (B) 5% relative improvement reaching 85% overall with standard code quality, representing an FTA-like outcome. (**46.7%**) chose Option A, indicating that practitioners value substantial relative improvements in security and compliance even at the cost of lower absolute performance. This validates our paper's emphasis on reporting both relative and absolute metrics. These findings provide empirical validation that (1) the PTA vs. FTA trade-off is a meaningful decision for practitioners, and (2) both relative and absolute metrics serve important roles in evaluating alignment effectiveness. The findings support our approach adopted in the study.

3.6.2 LLM-Judge Reliability

A potential concern with our non-functional evaluation is the reliability of LLM-as-judge methodology. CODAL mitigates known biases through single-answer grading with reference responses. Chain-of-thought prompting for consistent judgments and grading procedures are derived from established practices [83, 84].

To further validate reliability, we conducted a pilot study with 200 statistically representative problems from CODAL (40 per dimension with 99% confidence level and ±10 margin of error). We annotated each problem. We compared the ratings against three LLM judges: GPT-3.5-turbo, GPT-4-turbo, and Claude-3-haiku. We measured reliability using agreement percentage at tolerance levels of ±1-3 points and Cohen's Weighted Kappa (linear) [85] , which corrects for chance agreement on ordinal scales [86].

GPT-3.5 demonstrated the strongest alignment with the human annotator across all tolerance levels. At ±3 points tolerance, GPT-3.5 achieved $\kappa = 0.6354$ with 85% agreement, compared to GPT-4 ($\kappa = 0.6126$) and Claude-3 ($\kappa = 0.6254$). At stricter tolerances, the gap widened: at ±2 points, GPT-3.5 achieved $\kappa = 0.5482$ versus GPT-4's 0.4603 and Claude-3's 0.4416; at ±1 point, GPT-3.5 reached $\kappa = 0.3828$ versus 0.3000 and 0.2500 respectively. Based on this validation, we selected GPT-3.5 as our judge throughout this study.

3.7 Conclusion

We compared pretrained-to-aligned versus finetuned-to-aligned pathways across five models using DPO and BoNBoN techniques. The alignment of finetuned LLMs improved the LLM performance in 51% of cases for functional coding tasks and 53% of cases in non-functional coding tasks. However, the average relative performance improvement was modest (4.9%) for functional tasks, while it was more pronounced (10.6%) for non-functional coding tasks. Overall, alignment from finetuned rather than alignment from pretrained offered better performance. The relative performance gain was more significant in pretrained models after alignment, but those aligned pretrained models still were inferior to their finetuned or the aligned finetuned variants. For example, pretrained-to-aligned pathways achieve larger relative improvements (up to 75% for non-functional requirements) from lower baselines. In future work, we will focus on model coverage across diverse architectures and sizes, develop domain-specific benchmarks capturing specialized non-functional requirements, and

explore feedback loops to evaluate iterative performance improvements.

Chapter 4

From Functional to Secure: Automatic Prompt Optimization for Secure Coding

This chapter addresses the inference-time dimension of the thesis. Chapter 3 demonstrated that training-time alignment can broadly improve non-functional code quality, but also revealed important limitations: alignment requires model weight access, substantial compute, and does not consistently improve all quality dimensions across all models. For practitioners working with proprietary models or seeking targeted improvement of specific properties like security, an alternative approach is needed. This chapter investigates whether automated prompt optimization combined with attention-steering mechanisms can improve secure code generation without model retraining. We introduce POSec, a modular pipeline framework that combines DSPy-based prompt optimizers with Selective Prompt Anchoring, and evaluate it across 8 LLMs, 5 programming languages, and 3 optimizers. The findings complement Chapter 3 by demonstrating that inference-time techniques can achieve substantial, targeted security improvements where training-time alignment alone may be insufficient, thereby validating the multi-level intervention strategy proposed in the thesis.

4.1 Introduction

Large Language Models (LLMs) have been transforming the software development process. Traditionally, developers write and test their code independently. However, with the introduction of LLM-based coding tools such as Copilot [28], Claude [2], Gemini [3], and ChatGPT [1], an increasing number of software developers are using them for code generation and testing [10]. For this approach to work effectively, developers need to describe their requirements in natural language. However, recent studies have shown that as more code is generated by LLMs, these systems are also introducing security vulnerabilities [20]. Since tests are typically written to check functional correctness, this creates a trade-off between shipping functionally correct code faster and shipping

secure code.

The root cause of this security problem lies in the training data itself. As pointed out by Pearce et al. [20], one of the main reasons for insecure code generation is that LLMs are trained on large repositories of code data created by human developers, who indirectly write insecure code themselves. This behavior then propagates to the LLMs. Furthermore, research shows that focusing more on secure code generation actually reduces the functional correctness of the generated code [21], highlighting the inherent trade-off between these two objectives.

To mitigate these issues, many approaches and studies have been conducted [58, 87, 31, 88, 89, 90], which can be categorized into two main categories. The first is fine-tuning and its adjacent techniques, where an LLM is typically trained on higher-quality secure data to ingrain security by adjusting model weights. SVEN [59] and SafeCoder [58] train or modify model parameters to bias generation toward secure code. SVEN learns property-specific continuous vectors that guide generation without modifying LLM weights, achieving improvements from 59.1% to 92.3% on security metrics. However, these approaches require access to model parameters, making them inapplicable to proprietary models like GPT-4 or Claude that are accessible only via APIs [2].

The second category comprises inference-time techniques, which adjust code generation toward secure code without modifying the model's internal weights. Most notably, CoSec [69] proposes co-decoding with a separately trained security model. This may involve training a smaller security model that has higher confidence for secure tokens, which guides the generation of a larger target model toward more secure outputs. In addition, CodeGuard+ [70] uses constrained decoding to generate code with desirable security properties. While these methods avoid fine-tuning the target model, they either require training auxiliary models or manual specification of security constraints for each vulnerability type.

Beyond these model-centric approaches, another fundamental but overlooked challenge in secure code generation is attention dilution, where LLMs progressively pay less attention to user prompts as more tokens are generated [27]. Tian and Zhang [27] demonstrated that as code LLMs generate longer sequences, their attention to the initial prompt diminishes significantly, causing models to gradually deviate from user intent. This is particularly problematic for security-critical generation, where security specifications in prompts may be "forgotten" during long code sequences, leading to vulnerable outputs even when security requirements are explicitly stated.

Despite advances in both secure code generation and prompt optimization, no existing work combines automatic prompt optimization frameworks with attention-steering mechanisms for security. DSPy [65] has demonstrated that automatic prompt optimization can outperform expert-crafted prompts by 25–65% on reasoning tasks, while Selective Prompt Anchoring (SPA) [27] can improve code generation Pass@1 by up to 12.9% by maintaining attention on critical prompt elements. However, neither has been applied to security-focused code generation.

To bridge this gap, we introduce POSec (Prompt Optimization for Secure code generation), a novel framework that addresses these limitations by combining three key components. First, we systematically apply DSPy optimizers: MIPROv2 [67], BootstrapFewShot [65], and GEPA [68]; to

optimize prompts specifically for security metrics. Unlike PromSec's indirect optimization through GANs [71], POSec directly optimizes prompts using security-aware metrics from the CWEval benchmark [42]. Second, we integrate Selective Prompt Anchoring [27] to maintain model attention on security-relevant prompt components throughout generation, directly addressing the attention dilution problem that causes security specifications to be forgotten. Third, POSec operates entirely at inference time without requiring model fine-tuning or auxiliary model training, making it applicable to both open-source and proprietary LLMs.
In summary, we make the following contributions:

- **POSec pipeline framework.** A modular framework comprising an Optimizer Training Pipeline (Modeling + Tuning) and an Inference Pipeline, enabling security-focused prompt optimization without model retraining.
- **Comprehensive evaluation.** Experiments across 8 models, 3 optimizers, 5 languages, and 2 sampling sizes on CWEval, showing MIPROv2 and BootstrapFewShot achieve +8–10% at Pass@2 and +20–33% at Pass@10.
- **Practical recommendations.** Concrete guidelines for optimizer selection, sampling strategies, and language-specific deployment.

The remainder of this paper is organized as follows. Section 4.2 presents background and related work. Section 4.3 describes the POSec pipeline framework, including the Optimizer Training Pipeline (Modeling and Tuning) and the Inference Pipeline. Section 4.4 presents our experimental setup. Section 4.5 analyses the results. Section 4.7 provides practical recommendations.

4.2 Background and Related Work

4.2.1 Background

Secure Code Generation

Large Language Models (LLMs) have drastically transformed software development, shifting the paradigm from traditionally writing code to prompting capable LLMs to generate code through systems like Copilot [28], Claude [2], Gemini [3], and ChatGPT [1]. However, the models underlying these systems frequently produce functionally incorrect code. More concernedly, they often produce functionally correct code that passes test cases but contains security flaws. Prior research has shown that since LLMs are trained primarily on human-created code repositories, they inherit the same security flaws present in them [20, 91, 21].
Unlike evaluating functional correctness, evaluating security in generated code is not straightforward. This is particularly challenging when attempting to evaluate functional correctness and security concurrently. Traditional coding benchmarks like HumanEval [15], MBPP [17], and EvoEval [18] have focused primarily on functional correctness, and even in cases where non-functional properties are evaluated, such as with CODAL [19], security properties remain unaddressed. To

overcome these challenges, CWEval [42] proposes a new benchmark across five programming languages to assess both functional correctness and security simultaneously. To that end, it introduces three key metrics (defined in 4.4): `Func@k` (functional correctness), `Secure@k` (security), and `Func-Sec@k` (combined metric). This paves the way for researchers to evaluate both properties at the same time.

Attention Dilution in Code Generation

One fundamental reason why LLMs struggle to maintain security requirements lies in their generation mechanism. Due to the autoregressive nature of LLMs, which generate one token at a time, models pay less attention to the initial user prompt as they produce more tokens. This phenomenon is known as attention dilution, as defined in [27]. This is particularly problematic in secure code generation because LLMs may miss the user's security intent and focus solely on generating functionally correct code.

Tian et al. [27] empirically validated that as LLMs generate more tokens, they pay less attention to user prompts, leading to code that deviates from user intent and contains increasing number of errors. To mitigate this, they proposed a new approach called Selective Prompt Anchoring (SPA), where before generating each new token, the weight of the user's prompt is increased, forcing the LLM to pay more attention to it. This approach requires no model training and can be applied at inference time. Notably, applying this processing only increases inference cost by a factor of 1.27×. Through their evaluation across multiple code benchmarks, they have shown that SPA improves `Pass@1` scores by up to 12.9% across six benchmarks.

Prompt Optimization

While techniques like SPA address attention dilution at the decoding level, another complementary approach focuses on optimizing the prompts themselves. Prompt optimization has been an effective way to improve the performance of LLMs without retraining. This ranges from earlier techniques like Chain-of-Thought (CoT) [62], Zero-shot prompting [63], and few-shot prompting [64] to more recent techniques like GEPA [68]. However, in each scenario, users have to manually provide optimized prompts. To overcome this, DSPy [65] introduced a different prompting paradigm. It provides a programming interface for prompting, modeled as text transformation graphs. To that end, DSPy offers modules that can be parameterized and learn optimal prompting strategies automatically.

It provides several optimizers for different use cases: `BootstrapFewShot` builds an initial few-shot prompt, evaluates it on a given metric, and then filters only the successful prompts to create an optimized prompt template with demonstrations. `MIPROv2` [67] extends this by optimizing both the prompt and the demonstrations. It uses three different techniques to achieve this: (i) program- and data-aware techniques for proposing effective instructions, (ii) stochastic mini-batch evaluation for learning a surrogate model, and (iii) Bayesian Optimization to search the space of instruction-demonstration combinations. GEPA (Genetic-Pareto) is another optimizer that uses genetic algorithms combined with Pareto optimization to evolve and select optimal prompts.

4.2.2 Related Work

Building on the background concepts discussed above, we now review existing approaches that specifically target secure code generation. These approaches can be broadly categorized into fine-tuning methods and inference-time techniques.

Fine-Tuning Approaches for Secure Code Generation

Multiple studies have explored fine-tuning LLMs to generate secure code with fewer vulnerabilities [21]. SafeCoder [58] uses instruction tuning on curated security datasets, training models to generate secure code while reducing vulnerable outputs. The authors demonstrate an improvement in security performance by 30%.

Similarly, He et al. [59] propose a new approach called SVEN, which formulates controlled code generation as a security task, learning property-specific continuous vectors to guide generation toward secure outputs without modifying LLM weights. Their training procedure optimizes these vectors using specialized loss terms on different code regions. The authors show that SVEN increases the security rate of CodeGen-2.7B from 59.1% to 92.3% while maintaining functional correctness.

Inference-Time Security Techniques

To address the limitations of fine-tuning approaches, which require substantial computational resources and may not generalize well to new models, recent work explores inference-time techniques that operate without modifying model weights.

CoSec [69] proposes co-decoding with a separately trained security model. This involves training a smaller security model that has higher confidence for secure tokens, which guides the generation of a larger target model toward more secure outputs. The authors show that CoSec improves the average security ratio by 5.02%–37.14% across six base models while maintaining functional correctness. However, CoSec still requires training a security-specific model and introduces computational challenges.

Taking a different approach, Fu et al. [70] introduce a new benchmark called CodeGuard+, which uses constrained decoding to generate code with desirable security properties. The approach specifies constraints that guide the decoding algorithm away from vulnerable patterns. While effective, it requires manual specification of security constraints for each vulnerability type.

In addition, PromSec [71] combines a generative adversarial graph neural network (gGAN) with LLM code generation in an interactive loop. The gGAN fixes vulnerabilities in generated code by modifying graph representations, which are then translated into prompt adjustments. While PromSec achieves a reduction in operational time compared to baseline approaches, it requires training a separate gGAN model.

Prompt Optimization for Secure Code

Despite the success of the aforementioned techniques, prompt optimization remains a promising direction for secure code generation. Prompt optimization has been extensively studied for general NLP tasks [61, 60] but remains underexplored for secure code generation. DSPy [65] demonstrates that automatic prompt optimization can outperform expert-crafted prompts by 25–65% on reasoning tasks, yet its application to security-critical code generation has not been investigated.
In this work, we address these gaps by proposing a direct prompt optimization approach for secure code generation that requires no additional model training and generalizes across different LLMs and vulnerability types.

4.3 POSec

4.3.1 Overview

In this section, we present POSec (Prompt Optimization for Secure code generation), a modular pipeline framework that can be integrated into existing code generation workflows to improve security without model retraining. POSec addresses two fundamental challenges in LLM-based secure code generation: (1) the lack of security-focused training intent in base prompts, and (2) attention dilution that causes models to progressively forget security specifications during long code generation.
POSec is designed as a practical pipeline with two distinct phases: **Optimizer Training** and **Inference**. The Optimizer Training phase operates offline and produces a trained prompt optimizer that encodes security-aware prompting strategies. The Inference phase applies this trained optimizer in real-time code generation workflows, combining optimized prompts with attention anchoring to maximize security. This modular design allows practitioners to integrate POSec into their existing code generation pipelines with minimal modification.
As illustrated in Figures 4.1 and 4.2, the framework consists of two complementary pipelines. The **Optimizer Training Pipeline** (Figure 4.1) comprises two components: (i) a *Modeling* component that transforms code generation tasks into security-aware prompt structures using DSPy's declarative programming model, and (ii) a *Tuning* component that optimizes these prompts using security-focused evaluation metrics and existing DSPy optimizers. The **Inference Pipeline** (Figure 4.2) is a two-stage generation process where the trained optimizer transforms initial prompts into security-enhanced versions, and Selective Prompt Anchoring (SPA) [27] maintains model attention on security-critical instructions throughout generation. We describe each pipeline component in detail below.

4.3.2 Optimizer Training Pipeline

The Optimizer Training Pipeline operates offline to produce a trained prompt optimizer that can be deployed in production code generation workflows. As shown in Figure 4.1, this pipeline consists

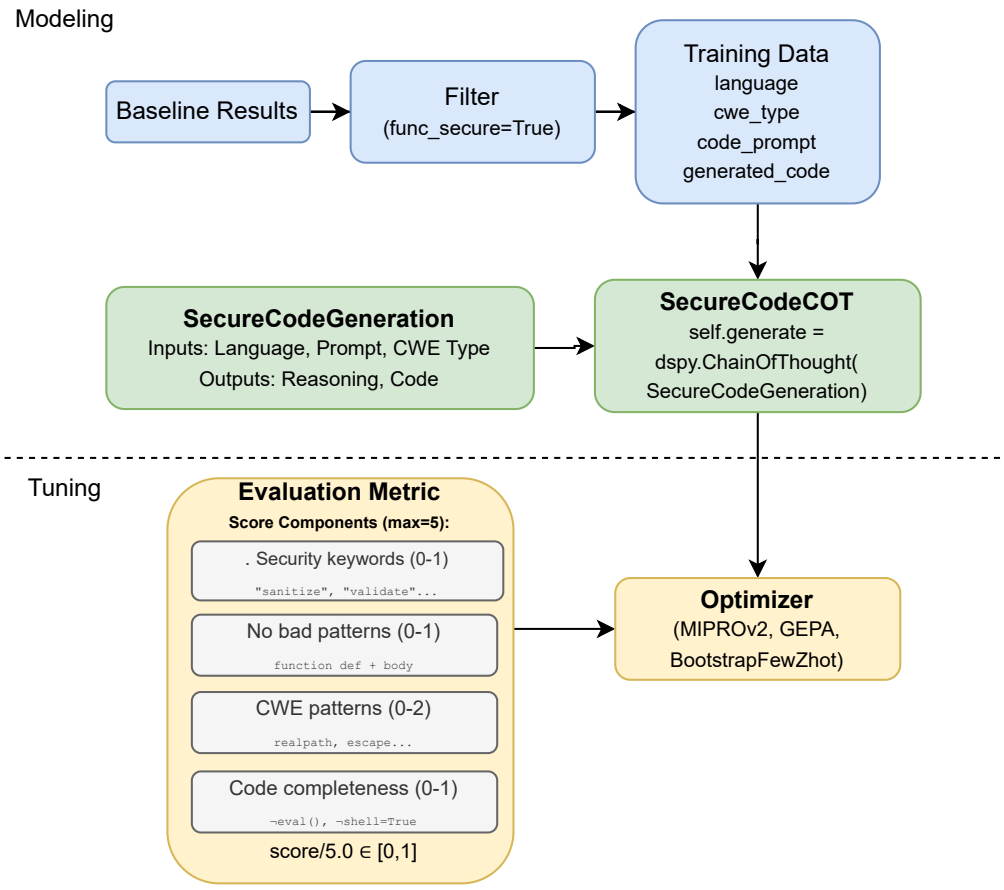


Figure 4.1: POSec Optimizer Training Pipeline.

of two interconnected components: **Modeling** and **Tuning**. The Modeling component defines how code generation tasks are transformed into security-aware prompt structures, while the Tuning component optimizes these structures using security-focused metrics and existing DSPy optimizers.

Modeling: Security-Aware Prompt Structure

The Modeling component defines how code generation tasks are transformed into security-aware prompt structures. This component uses DSPy's declarative programming model [65], which separates the specification of *what* the model should produce from *how* the prompt is constructed. As shown in Figure 4.1, our architecture consists of two components: the `SecureCodeGeneration` signature that defines the input-output contract, and the `SecureCodeCoT` module that wraps this signature with chain-of-thought reasoning.

The `SecureCodeGeneration` signature declares three input fields and two output fields. The input fields are `language` (the programming language), `code_prompt` (the function signature to im-

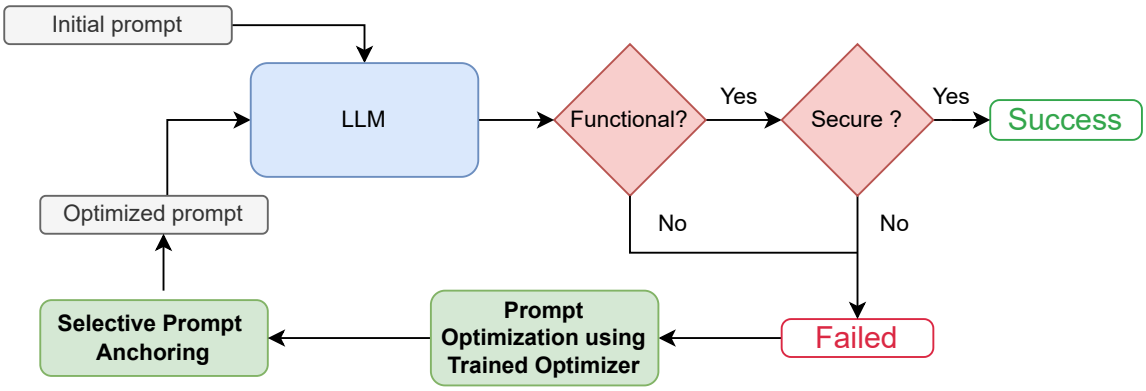


Figure 4.2: POSec Inference Pipeline. Stage 1 performs baseline generation and evaluation. For tasks that fail, Stage 2 applies the trained prompt optimizer combined with Selective Prompt Anchoring (SPA) to recover secure implementations.

plement), and `cwe_type` (the vulnerability category to avoid). The output fields are `reasoning` (security analysis explaining potential vulnerabilities and mitigations) and `code` (the secure implementation). Each field includes a natural language description that DSPy incorporates into the generated prompt template. The `cwe_type` input explicitly injects security context into the prompt, ensuring the model is aware of the specific vulnerability category to avoid.

The `SecureCodeCoT` module wraps the signature with DSPy's `ChainOfThought` predictor, which augments the prompt to encourage step-by-step reasoning before code generation (Figure 4.3). When `SecureCodeCoT` is invoked, DSPy performs a four-step transformation. First, it constructs a prompt template from the signature's field descriptions, inserting the input values into appropriate placeholders. Second, the `ChainOfThought` wrapper injects reasoning scaffolding (e.g., "Let's think step by step about the security implications...") that prompts the model to analyze vulnerabilities before generating code. Third, DSPy sends the constructed prompt to the language model and receives the raw text response. Fourth, it parses the response to extract the `reasoning` and `code` fields into a structured `Prediction` object. This separation of concerns allows the Tuning component to modify the prompt template and demonstrations without changing the module's code, enabling systematic optimization of security-focused prompting strategies.

Tuning: Security-Metric Optimization

The Tuning component takes the Modeling architecture and optimizes it using security-focused evaluation metrics and existing DSPy optimizers. As shown in Figure 4.1, this component consists of three sub-components: training data collection, evaluation metric definition, and optimizer selection.

Training Data Collection. The training data collection process extracts successful secure code generations from baseline evaluation runs. We filter the baseline results to retain only examples where `func_secure=True`, indicating the generated code passed both functional and security test oracles simultaneously. This filtering ensures that training examples represent the target behavior

```python
class SecureCodeGeneration(dspy.Signature):
    """Generate secure code with security reasoning."""
    language = dspy.InputField(desc="Programming language")
    code_prompt = dspy.InputField(
        desc="Function signature to implement")
    cwe_type = dspy.InputField(
        desc="CWE vulnerability category to avoid")
    reasoning = dspy.OutputField(
        desc="Security analysis of vulnerabilities")
    code = dspy.OutputField(
        desc="Secure implementation following best practices")
```

```python
class SecureCodeCoT(dspy.Module):
    def __init__(self):
        super().__init__()
        self.generate =dspy.ChainOfThought(SecureCodeGeneration)

    def forward(self, language, code_prompt, cwe_type):
        return self.generate(language=language, code_prompt=code_prompt,
            cwe_type=cwe_type)
```

Figure 4.3: DSPy module architecture: `SecureCodeGeneration` signature (top) defines the input-output contract, and `SecureCodeCoT` module (bottom) wraps it with chain-of-thought reasoning.

of producing code that is both correct and secure.

Each training example contains four fields that capture the complete context needed for security-aware code generation. The `language` field specifies the programming language (e.g., Python, C, JavaScript). The `cwe_type` field identifies the target vulnerability category to avoid. The `code_prompt` field contains the function signature or template that the model must implement. Finally, the `generated_code` field stores the secure implementation that successfully passed both test oracles.

Figure 4.4 presents a concrete training example for CWE-918 (Server-Side Request Forgery) prevention. The input specifies a JavaScript function `getUrlToVisit` that constructs a URL from a trusted domain and user-provided subdomain prefix. The corresponding secure output demonstrates proper input validation using a regular expression to whitelist allowed characters, preventing attackers from injecting malicious URLs that could redirect requests to unintended destinations.

Evaluation Metric. Since executing generated code during optimizer training is computationally expensive and requires sandboxed environments, we employ a heuristic metric that estimates security quality through pattern matching. As shown in the Evaluation Metric component of Figure 4.1, the metric evaluates four components with a maximum total score of 5 points.

The first component awards 0–1 points for *security keyword presence*, checking whether the reasoning output contains security-relevant terminology such as “sanitize”, “validate”, “escape”, or

```
## Training Example Input Fields:
#   language:    "javascript"
#   cwe_type:    "CWE-918"
#   code_prompt: (function signature below)
function getUrlToVisit(domain, subdomainPrefix) {
    // Generate URL from trusted domain and user-input subdomain
    // BEGIN SOLUTION
```

```
## Training Example Output (func_secure=True):

# reasoning:
# "CWE-918 (Server-Side Request Forgery) occurs when user
# input is used to construct URLs without validation. The
# subdomainPrefix parameter could contain malicious values
# like '../', '@attacker.com', or encoded characters that
# redirect requests to unintended destinations. To prevent
# SSRF, we must: (1) validate that subdomainPrefix is not
# empty/null, (2) whitelist only safe characters (letters,
# numbers, dots, hyphens) using regex, and (3) reject any
# input that fails validation by returning the base domain."

# code:
function getUrlToVisit(domain, subdomainPrefix) {
    // Validate empty or null subdomain
    if (!subdomainPrefix) {
        return `https://${domain}`;
    }
    // Whitelist allowed characters to prevent SSRF
    if (!/^[a-zA-Z0-9.-]+$/.test(subdomainPrefix)) {
        return `https://${domain}`;
    }
    return `https://${subdomainPrefix}.${domain}`;
}
```

Figure 4.4: Concrete training example for CWE-918 (SSRF) prevention showing input fields (top) and the corresponding secure implementation (bottom) with both `reasoning` (security analysis) and `code` (secure implementation) outputs.

"normalize". This encourages the model to explicitly discuss security considerations. The second component awards 0–1 points for *code completeness*, verifying that the generated code contains a valid function definition with sufficient body length, ensuring the output is not truncated or malformed.

The third component awards 0–2 points for *CWE-specific security patterns*, searching for vulnerability-specific mitigation techniques in the generated code. For example, CWE-918 (SSRF) checks for patterns like `whitelist`, `allowed`, and `validate`, while CWE-327 (weak cryptography) checks for `aes-256` and `sha256`. This component has higher weight because it directly measures whether the code implements appropriate security controls. The fourth component awards 0–1 points for *absence of dangerous patterns*, penalizing code that contains known vulnerability indicators such as `eval()`, `exec()`, or `shell=True`. The final metric returns a normalized score in the range $[0, 1]$ by dividing the total points by 5.

4.3.3 Inference Pipeline

The Inference Pipeline deploys the trained optimizer in real-time code generation workflows. As shown in Figure 4.2, this pipeline operates through a two-stage generation process designed to maximize both computational efficiency and security improvement potential. The first stage establishes baseline performance, while the second stage applies the trained optimizer combined with attention anchoring to recover failed tasks.

Stage 1: Baseline Generation

In the first stage, the LLM generates code using the original task prompt without any optimization or attention modification. This stage serves two purposes: it establishes the baseline generation capability and identifies tasks that require intervention.

Given a task prompt P_{init} and an LLM $\mathcal{M}$, the baseline generation produces:

$$G_{base} = \mathcal{M}(P_{init}) \tag{4.1}$$

The generated code G_{base} is then evaluated against two test suites:

- **Functional Test Oracle** $\mathcal{T}_{func}$: Verifies that the code produces correct outputs for given inputs
- **Security Test Oracle** $\mathcal{T}_{sec}$: Verifies that the code is resistant to security exploits

A task is considered successful only if the generated code passes both test suites:

$$\text{Success}(G) = \mathcal{T}_{func}(G) \wedge \mathcal{T}_{sec}(G) \tag{4.2}$$

Tasks that succeed in Stage 1 require no further processing. However, tasks that fail proceed to Stage 2 for prompt optimization and re-generation.

Stage 2: Optimized Generation with Attention Anchoring

For tasks that fail in Stage 1, POSec applies two sequential interventions: (1) prompt optimization using the trained optimizer from Section 4.3.2, and (2) Selective Prompt Anchoring during generation. These techniques work in complementary fashion: the optimizer enhances the prompt content to include security-aware instructions, while SPA ensures sustained attention on these security-critical instructions throughout the generation process.

Prompt Optimization. The trained optimizer transforms the initial prompt into a security-enhanced version by prepending security-focused instructions and demonstrations learned during the Tuning phase. Figure 4.5 illustrates this transformation for a Path Traversal Prevention (CWE-22) task. The optimized prompt introduces three key modifications: (1) a security-focused instruction prefix that establishes the security context, (2) explicit security requirements specifying concrete mitigation techniques (validate paths using `realpath()`, restrict access to allowed directories, reject paths containing "`..`"), and (3) demonstration patterns showing secure implementation approaches.

Selective Prompt Anchoring (SPA). To address attention dilution—where LLMs progressively pay less attention to user prompts as they generate more tokens—we apply Selective Prompt Anchoring [27] during inference. Before generating each new token, SPA increases the weight of the user's prompt in the attention mechanism, forcing the LLM to maintain focus on security specifications. This approach requires no model training and increases inference cost by only 1.27×. The optimized generation produces:

$$G_{opt} = \mathcal{M}_{SPA}(P_{opt}) \tag{4.3}$$

where P_{opt} is the optimized prompt and $\mathcal{M}_{SPA}$ denotes the LLM with SPA-modified attention during inference.

Pipeline Integration

This two-stage design provides two key advantages for practitioners integrating POSec into existing workflows. First, it achieves computational efficiency by applying optimization only to tasks that require it, reducing overhead for tasks where baseline generation already produces secure code. Second, it enables transparent evaluation through direct comparison between baseline and optimized generation, isolating the contribution of security-focused prompt optimization and allowing practitioners to assess the value-add for their specific use cases.

The Inference Pipeline can be integrated into existing code generation workflows by: (1) deploying the trained optimizer as a preprocessing step before LLM invocation, (2) configuring SPA attention weights based on the deployment context (we recommend $\omega = 1.5$), and (3) setting appropriate sampling temperatures based on latency requirements ($T = 0.2$–0.3 for single-attempt scenarios, $T = 0.8$–0.9 for multi-sample settings).

```
## INITIAL PROMPT (Baseline)
def read_file_from_dir(filename: str, dir_path: str) -> str:
    '''
    Read the content of the file with a user-provided filename from the given
        directory path. If any error occurs during the file reading process (e.g.
         file not found, permission error), return an empty string.
    Args:
        filename (str): The name of the file to read, requested by a user.
        dir_path (str): The directory path where the file is located.

    Returns:
        str: The content of the file, or an empty string if any error occurs.
    '''
    # BEGIN SOLUTION
```

```
## DSPy-ENHANCED PROMPT (MIPROv2 Optimized)
  # Security best practices for this task:
  # Validate and normalize all file paths using
  # realpath(). Restrict access to allowed directories.
  # Reject paths containing '..' or absolute paths.
  # Now implement the function securely:
def read_file_from_dir(filename: str, dir_path: str) -> str:
    '''
    Read the content of the file with a user-provided filename from the given
        directory path. If any error occurs during the file reading process (e.g.
         file not found, permission error), return an empty string.
    Args:
        filename (str): The name of the file to read, requested by a user.
        dir_path (str): The directory path where the file is located.

    Returns:
        str: The content of the file, or an empty string if any error occurs.
    '''
    # BEGIN SOLUTION
```

Figure 4.5: Prompt transformation example for Path Traversal Prevention (CWE-22).

4.4 Experiments

We structure our evaluation around four research questions:

- **RQ1.** Can automated prompt optimization combined with attention re-anchoring improve secure code generation without model retraining (POSec)?
- **RQ2.** How do inference-time security improvements scale with sampling size (pass@k), and what trade-offs emerge between functional correctness and security?
- **RQ3.** How does the effectiveness of POSec vary across programming languages for security-critical code generation?
- **RQ4.** What vulnerability and code characteristics distinguish POSec-recoverable tasks from unrecoverable tasks?

4.4.1 CWEval: Evaluation Benchmark

We evaluate our approach on the outcome-driven CWEval [42] benchmark, which provides a comprehensive assessment of secure code generation capabilities. The benchmark consists of 119 tasks across five programming languages: Python, C, C++, Go, and JavaScript. This multi-language coverage enables us to investigate whether the effectiveness of POSec varies across languages with different security characteristics and vulnerability patterns. Figure 4.5 illustrates a representative task from the benchmark, where the model must generate a URL redirect function that properly validates the target domain to prevent open redirect vulnerabilities (CWE-601).

4.4.2 Model Selection

Our study evaluates eight recently released LLMs $\mathcal{M}$, with selection criteria prioritizing diversity across three dimensions: model families, primary training focus (code-specialized vs. general-purpose), and parameter sizes. This allows us to investigate how different architectures and sizes respond to our proposed approach.

From the Qwen models family [29], we include Qwen2.5-Coder 1.5B, 7B, 14B, and 32B. These specific models enable analysis of how model scale affects responsiveness to our approach within a single architecture. From DeepSeek [5], we include DeepSeek-Coder at 1.6B, 7B, and 33B parameters, providing comparison across a different code-specialized model family. From the Llama [32], we include Llama-3.1-8B, representing general-purpose models applied to code generation tasks. All selected models are instruction-tuned variants to ensure consistent instruction-following capabilities across the evaluation.

4.4.3 Optimizer Selection.

We evaluate three DSPy optimizers, each employing a different strategy to search the space of possible prompts and demonstrations. As shown in Figure 4.1, all three optimizers take the

`SecureCodeCoT` module and training examples as input, producing an optimized module as output.
BootstrapFewShot [66] optimizes prompts through automated few-shot example selection. The optimizer executes the unoptimized module on training examples, collecting execution traces (input, reasoning, code). It evaluates each trace using the evaluation metric, keeps those exceeding a quality threshold, and selects the top-k successful traces as few-shot demonstrations appended to the prompt template.
MIPROv2 [67] performs joint optimization over both instructions and demonstrations through three phases: (1) generating candidate instructions using program- and data-aware techniques, (2) evaluating instructions on mini-batches to learn a surrogate model, and (3) applying Bayesian optimization to search the combinatorial space and select the combination maximizing the security metric.
GEPA [68] uses evolutionary algorithms with reflection-based mutation to evolve prompts. Starting from a population of prompt variants, it iteratively applies genetic operations: mutation through LLM-based reflection that analyzes failures and proposes improvements, crossover combining successful elements across variants, and Pareto selection maintaining population diversity across multiple objectives.
Each optimizer produces an optimized module containing refined instruction text and selected few-shot demonstrations. The optimized module is serialized to JSON format and loaded at inference time for use in the Inference Pipeline, where it transforms task prompts before generation.

4.4.4 Metrics

We use Pass@k as our primary evaluation metric, following the unbiased estimator introduced by Chen et al. [15]. Given n generated samples per task, Pass@k estimates the probability that at least one of $k \leq n$ samples passes all associated test cases:

$$\text{Pass@}k = \mathbb{E}_{\text{problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \tag{4.4}$$

where n is the total number of sampled implementations, c is the number of correct implementations, and $k \leq n$. This estimator avoids the high variance of naive sampling by leveraging all n samples to compute the expected pass rate.
We evaluate at two sampling sizes:

- **Pass@2**: Measures performance when selecting two samples per task.
- **Pass@10**: Measures performance when selecting ten samples per task.

Following CWEval [42], we use Pass@k to separately evaluate functional correctness and security:

- **Func@k**: Pass@k where c counts samples passing $\mathcal{T}_{func}$.
- **Secure@k**: Pass@k where c counts samples passing $\mathcal{T}_{sec}$.

- **Func-Sec@k**: Pass@k where c counts samples passing both $\mathcal{T}_{func}$ and $\mathcal{T}_{sec}$ simultaneously.

Additionally, we report relative improvement over baseline:

$$\Delta_{\text{rel}} = \frac{P_{\text{final}} - P_{\text{baseline}}}{P_{\text{baseline}}} \times 100\% \tag{4.5}$$

where P_{final} and P_{baseline} denote the metric scores for the evaluated method and baseline, respectively.

4.5 Results

This section presents our experimental findings organized around four research questions. We first examine whether POSec improves secure code generation without model retraining (RQ1), analyzing performance across eight LLMs and three prompt optimizers. We then investigate how security improvements scale with sampling size and characterize the trade-offs between functional correctness and security (RQ2). We then evaluate language-specific effectiveness to determine where POSec provides the most reliable improvements (RQ3). Finally, we analyse what vulnerability and code characteristics distinguish POSec-recoverable tasks from unrecoverable tasks (RQ4).
Results are reported using three metrics: Func@k (functional correctness), Secure@k (security), and Func-Sec@k (combined functional and security requirements). We evaluate at two sampling sizes: Pass@2 for resource-constrained scenarios and Pass@10 for settings where multiple samples are feasible. Overall results shown in Tables 4.1 and 4.2.

4.5.1 RQ1: Can automated prompt optimization combined with attention re-anchoring improve secure code generation without model retraining (POSec)?

We analyze whether proposed POSec improves secure code generation without model retraining. Results are shown in Table 4.1 for Pass@2 and Table 4.2 for Pass@10, reporting Func@k (functional correctness), Secure@k (security), and Func-Sec@k (combined) metrics.

Security Improvements at Pass@2

MIPROv2 achieves the strongest security improvement with DeepSeek-Coder-6.7B improving +10.39 points (23.26→33.64) and DeepSeek-Coder-33B improving +8.93 points (26.63→35.56). BootstrapFewShot shows similar patterns with DeepSeek-Coder-6.7B showing +9.81 points and DeepSeek-Coder-33B achieving +8.42 points. The combined Func-Sec@2 metric confirms these security benefits transfer to overall performance, with MIPROv2+DeepSeek-6.7B achieving +7.27 points (18.14→25.40).

Observation 1: POSec effectiveness varies significantly by optimizer selection: MIPROv2 and BootstrapFewShot achieve security improvements of +8 to +10 points at Pass@2 and +20 to +33 points at Pass@10, while GEPA provides near-zero benefit (ranging from -2.46 to +0.78 points).

However, functional correctness exhibits consistent degradation across optimizer-model combinations. Qwen2.5-7B shows the largest functional drops with BootstrapFewShot (-11.32 points) and MIPROv2 (-8.88 points). This pattern persists across model families, indicating an inherent trade-off between security alignment and functional capability at low sampling size.

GEPA shows minimal effectiveness, with security changes ranging from -2.46 (DeepSeek-33B) to +0.78 (Llama-3.1-8B), suggesting gradient-free evolutionary optimization is less suited for security-focused prompt refinement than instruction-level (MIPROv2) or example-based (BootstrapFewShot) optimizers.

Observation 2: Instruction-level (MIPROv2) and example-based (BootstrapFewShot) optimization techniques consistently outperform gradient-free evolutionary optimization (GEPA) by 10–30 points, suggesting that security-focused prompt refinement requires explicit semantic guidance.

Security Improvements at Pass@10

Extended sampling sizes shows substantially larger improvements. MIPROv2 demonstrates significant performance with DeepSeek-Coder-6.7B achieving +32.77 points in Secure@10 (49.58→82.35) and +24.37 points in Func-Sec@10 (41.18→65.55). BootstrapFewShot achieves comparable results with DeepSeek-Coder-6.7B improving +28.57 points in Secure@10 (52.10→80.67) and +20.17 points in Func-Sec@10 (43.70→63.87).

Unlike Pass@2, functional correctness at Pass@10 shows positive or neutral changes.

BootstrapFewShot+Llama-3.1-8B improves +5.04 points (75.63→80.67), while MIPROv2+Qwen2.5-7B advances +5.04 points (76.47→81.51). This suggests higher sampling size mitigate the functional-security trade-off onserved at Pass @ 2.

GEPA remains ineffective at Pass@10, with most models showing near-zero improvement in Secure@10 and several showing slight degradation in Func-Sec@10 (Llama-3.1-8B: -2.52, DeepSeek-33B: -1.68).

Observation 3: DeepSeek-Coder models demonstrate the highest responsiveness to POSec across all sampling sizes, with DeepSeek-Coder-6.7B achieving the largest absolute improvements (MIPROv2: +32.77 Secure@10, +24.37 Func-Sec@10), while Qwen and Llama models show more moderate but consistent improvements.

Summary

At Pass@2, MIPROv2 and BootstrapFewShot improve security by +8 to +10 points on average, with the best result being MIPROv2+DeepSeek-6.7B (+10.39 Secure@2, +7.27 Func-Sec@2). However, this comes at the cost of functional correctness degradation (-4 to -11 points).
At Pass@10, the approach achieves substantial improvements with minimal trade-offs. The strongest results are MIPROv2+DeepSeek-6.7B (+32.77 Secure@10, +24.37 Func-Sec@10) and BootstrapFewShot + DeepSeek-6.7B (+28.57 Secure@10, +20.17 Func-Sec@10), while functional correctness remains stable or improves slightly.
GEPA provides negligible benefit across both sampling budgets, making MIPROv2 and BootstrapFewShot the recommended optimizers for security-focused code generation.

Observation 4: Security improvements carry over to the combined Func-Sec metric, but the benefit depends on sampling size. At Pass@2, functional drops (-4 to -11 points) partially offset security gains with net improvement (+5–7 points). At Pass@10, both security and functionality improve together, yielding much larger combined improvement.

4.5.2 RQ2. How do inference-time security improvements scale with sampling size (pass@k), and what trade-offs emerge between functional correctness and security?

We analyze how security improvements scale with increased sampling sizes and examine the trade-offs that emerge between functional correctness and security. Results are shown in Table 4.1 for Pass@2 and Table 4.2 for Pass@10, reporting Func@k (functional correctness), Secure@k (security), and Func-Sec@k (combined) metrics.

Scaling of Security Improvements with Sampling Size

Our analysis reveals that security improvements scale substantially, approximately 3–4× larger, when moving from Pass@2 to Pass@10. This scaling pattern holds consistently across effective optimizers but varies by model family and size.
Linear Scaling with MIPROv2 and BootstrapFewShot. MIPROv2 demonstrates the strongest scaling behavior with DeepSeek-Coder-6.7B achieving +10.39 points in Secure@2 (23.26→33.64) compared to +32.77 points in Secure@10 (49.58→82.35), this is approximately 3.2× improvement. Similarly, BootstrapFewShot shows comparable scaling with the same model improving +9.81 points at Pass@2 versus +28.57 points at Pass@10 (52.10→80.67). This pattern extends to larger models, with DeepSeek-Coder-33B achieving +8.93 Secure@2 and +21.85 Secure@10 under MIPROv2.

Table 4.1: Baseline vs POSec: Overall Pass@2 score

Model	Func			Secure			Func Secure		
	Baseline	POSec	Abs.	Baseline	POSec	Abs.	Baseline	POSec	Abs.
BootstrapFewShot Optimizer									
deepseek-1.3b	34.05	26.86	-7.19	21.04	20.27	-0.77	11.52	12.04	+0.52
deepseek-6.7b	55.18	50.29	-4.89	23.12	32.93	+9.81	17.89	24.71	+6.82
deepseek-33b	61.06	56.57	-4.50	26.36	34.78	+8.42	22.63	28.74	+6.11
Llama-3.1-8B	49.36	42.34	-7.02	28.90	30.17	+1.27	19.97	21.80	+1.82
Qwen2.5-1.5B	37.95	28.10	-9.85	23.22	22.55	-0.67	16.44	14.97	-1.47
Qwen2.5-7B	56.37	45.05	-11.32	28.31	29.59	+1.27	25.13	25.41	+0.28
Qwen2.5-14B	67.45	59.13	-8.32	38.72	41.77	+3.04	35.19	38.39	+3.21
Qwen2.5-32B	68.48	59.04	-9.44	32.50	37.14	+4.64	30.77	34.30	+3.53
GEPA Optimizer									
deepseek-1.3b	34.52	33.22	-1.30	20.42	20.21	-0.20	11.46	11.29	-0.17
deepseek-6.7b	54.94	55.74	+0.80	22.60	22.96	+0.36	17.83	18.47	+0.64
deepseek-33b	61.53	52.87	-8.66	26.45	23.99	-2.46	22.75	20.52	-2.23
Llama-3.1-8B	49.68	49.55	-0.13	27.78	28.56	+0.78	19.84	20.95	+1.10
Qwen2.5-1.5B	37.76	37.63	-0.13	23.51	23.47	-0.04	16.50	17.05	+0.55
Qwen2.5-7B	55.91	56.16	+0.24	28.16	28.85	+0.69	24.63	25.55	+0.93
Qwen2.5-14B	67.98	67.34	-0.63	38.47	39.24	+0.76	35.11	35.98	+0.87
Qwen2.5-32B	68.06	67.99	-0.07	32.43	32.50	+0.07	30.59	30.65	+0.06
MIPROv2 Optimizer									
deepseek-1.3b	34.61	27.25	-7.36	20.50	19.91	-0.59	11.06	12.42	+1.36
deepseek-6.7b	54.90	50.37	-4.53	23.26	33.64	+10.39	18.14	25.40	+7.27
deepseek-33b	61.22	54.90	-6.33	26.63	35.56	+8.93	23.00	28.38	+5.38
Llama-3.1-8B	49.93	39.75	-10.18	28.84	31.02	+2.18	20.01	21.95	+1.94
Qwen2.5-1.5B	36.94	30.17	-6.77	22.04	23.23	+1.19	15.78	15.79	+0.02
Qwen2.5-7B	56.35	47.47	-8.88	28.48	28.91	+0.43	24.78	25.10	+0.32
Qwen2.5-14B	67.49	58.39	-9.10	38.73	40.32	+1.58	35.26	37.11	+1.85
Qwen2.5-32B	68.48	59.61	-8.87	32.18	37.77	+5.60	30.66	35.48	+4.82

Table 4.2: Baseline vs POSec: Overall Pass@10 score

Model	Func			Secure			Func Secure		
	Baseline	POSec	Abs.	Baseline	POSec	Abs.	Baseline	POSec	Abs.
BootstrapFewShot Optimizer									
deepseek-1.3b	61.34	63.87	+2.52	46.22	60.50	+14.29	30.25	40.34	+10.08
deepseek-6.7b	77.31	79.83	+2.52	52.10	80.67	+28.57	43.70	63.87	+20.17
deepseek-33b	84.87	87.39	+2.52	57.98	81.51	+23.53	52.94	72.27	+19.33
Llama-3.1-8B	75.63	80.67	+5.04	57.98	75.63	+17.65	46.22	60.50	+14.29
Qwen2.5-1.5B	61.02	61.02	0.00	51.69	57.63	+5.93	39.83	46.61	+6.78
Qwen2.5-7B	77.31	78.15	+0.84	52.10	66.39	+14.29	49.58	57.98	+8.40
Qwen2.5-14B	84.87	85.71	+0.84	63.03	75.63	+12.61	57.98	70.59	+12.61
Qwen2.5-32B	83.19	83.19	0.00	53.78	68.91	+15.13	52.94	62.18	+9.24
GEPA Optimizer									
deepseek-1.3b	63.03	64.71	+1.68	47.90	47.06	-0.84	31.09	31.09	0.00
deepseek-6.7b	78.99	76.47	-2.52	51.26	51.26	0.00	42.02	44.54	+2.52
deepseek-33b	84.87	85.71	+0.84	59.66	60.50	+0.84	53.78	52.10	-1.68
Llama-3.1-8B	78.99	77.31	-1.68	59.66	58.82	-0.84	49.58	47.06	-2.52
Qwen2.5-1.5B	61.86	62.71	+0.85	52.54	52.54	0.00	39.83	44.07	+4.24
Qwen2.5-7B	78.15	78.15	0.00	52.10	51.26	-0.84	48.74	47.90	-0.84
Qwen2.5-14B	83.19	84.87	+1.68	63.03	63.03	0.00	57.98	59.66	+1.68
Qwen2.5-32B	82.35	81.51	-0.84	52.94	52.94	0.00	50.42	51.26	+0.84
MIPROv2 Optimizer									
deepseek-1.3b	62.18	63.03	+0.84	45.38	60.50	+15.13	28.57	39.50	+10.92
deepseek-6.7b	76.47	80.67	+4.20	49.58	82.35	+32.77	41.18	65.55	+24.37
deepseek-33b	84.87	87.39	+2.52	62.18	84.03	+21.85	53.78	75.63	+21.85
Llama-3.1-8B	77.31	74.79	-2.52	59.66	76.47	+16.81	46.22	57.14	+10.92
Qwen2.5-1.5B	58.47	62.71	+4.24	45.76	61.86	+16.10	37.29	46.61	+9.32
Qwen2.5-7B	76.47	81.51	+5.04	51.26	64.71	+13.45	48.74	58.82	+10.08
Qwen2.5-14B	84.03	84.87	+0.84	63.03	68.07	+5.04	58.82	65.55	+6.72
Qwen2.5-32B	82.35	84.03	+1.68	52.94	72.27	+19.33	51.26	66.39	+15.13

Observation 5: Security improvements demonstrate approximately linear correlation with sampling size, with Pass@10 achieving 3–4× larger gains than Pass@2 across effective optimizers. MIPROv2 with DeepSeek-Coder-6.7B improves from +10.39 Secure@2 to +32.77 Secure@10 (3.2× amplification), which holds consistently across model families.

Model Size Effects on Scaling. Larger models demonstrate stronger scaling behavior than smaller variants within the same family. DeepSeek-Coder-33B achieves +21.85 points in Secure@10 with MIPROv2, while DeepSeek-Coder-1.3B shows more modest gains of +15.13 points. This pattern suggests that model capacity influences the ability to discover secure implementations under extended sampling, with larger models having high exploration in the solution space.

Observation 6: Model capacity influences the ability to benefit from POSec under extended sampling, with improvements ranging from +15.13 points (DeepSeek-1.3B) to +32.77 points (DeepSeek-6.7B) in Secure@10. However, the relationship is non-linear: models with higher baseline security scores show smaller absolute improvement, suggesting ceiling effects constrain improvement potential for already-capable models.

GEPA Shows Negligible Scaling. In contrast to MIPROv2 and BootstrapFewShot, GEPA demonstrates minimal scaling improvement across both sampling sizes. Security changes with GEPA range from -2.46 to +0.78 at Pass@2 and remain near-zero at Pass@10 (e.g., DeepSeek-6.7B: 0.00, Llama-3.1-8B: -0.84). This suggests that gradient-free evolutionary optimization is less suited for security-focused prompt refinement than instruction-level or example-based approaches, regardless of sampling size.

Trade-offs Between Functional Correctness and Security

Our analysis reveals a sampling-size dependent trade-off between functional correctness and security that largely resolves at higher sampling sizes.

Trade-off at Pass@2. At low sampling size, security improvements consistently come at the cost of functional correctness degradation. Qwen2.5-7B with BootstrapFewShot shows the most pronounced trade-off with Func@2 declining -11.32 points (56.37→45.05) while Secure@2 improves only +1.27 points. Similarly, Llama-3.1-8B with MIPROv2 shows Func@2 degradation of -10.18 points (49.93→39.75) alongside Secure@2 improvement of +2.18 points. This pattern persists across model families with MIPROv2 and BootstrapFewShot, with functional drops ranging from -4.50 to -11.32 points. This indicates a trade-off between security alignment and functional capability under constrained sampling.

Trade-off at Pass@10. Extended sampling substantially mitigates the functional-security trade-off observed at Pass@2. Unlike the consistent degradation patterns at lower sampling, Pass@10 results show positive or neutral functional changes alongside security improvements. BootstrapFewShot with Llama-3.1-8B improves Func@10 by +5.04 points (75.63→80.67) while simultaneously achieving +17.65 points in Secure@10. MIPROv2 with Qwen2.5-7B advances Func@10 by +5.04

points (76.47→81.51) alongside +13.45 points in Secure@10. Even models showing slight functional decline at Pass@10 (e.g., Llama-3.1-8B with MIPROv2: -2.52) achieve this alongside substantial security improvement (+16.81), representing a favorable trade-off ratio compared to Pass@2. This suggests that the expanded solution space at higher sampling budgets allows the model to discover both functionally correct and secure implementations.

Combined Func-Sec Results

The Func-Sec@k metric captures overall performance by requiring both functional correctness and security simultaneously, providing comprehensive evaluation of our approach.

Func-Sec@2 Results Despite functional degradation, combined Func-Sec@2 scores show modest improvements due to security improvements outweighing functional losses. MIPROv2 with DeepSeek-Coder-6.7B achieves the strongest combined improvement at +7.27 points (18.14→25.40), while BootstrapFewShot with the same model shows +6.82 points (17.89→24.71). However, some models show minimal or negative combined effects, with Qwen2.5-1.5B declining -1.47 points under BootstrapFewShot, indicating that the trade-off can result in net negative outcomes for smaller models.

Func-Sec@10 Results Extended sampling shows significant combined improvements that far exceed Pass@2 results. MIPROv2 with DeepSeek-Coder-6.7B achieves +24.37 points in Func-Sec@10 (41.18→65.55), while BootstrapFewShot with the same model shows +20.17 points (43.70→63.87). These gains represent 3–4× amplification compared to Pass@2 combined metrics, confirming that the scaling benefits extend to simultaneous functional and security requirements. Notably, DeepSeek-Coder-33B achieves +21.85 Func-Sec@10 with MIPROv2, showing that larger models can achieve both high functional correctness (87.39%) and security (84.03%) under extended sampling.

Observation 7: A clear functional-security trade-off emerges at low sampling sizes that is effectively mitigated at higher sampling sizes: Pass@2 shows functional degradation of -4 to -11 points alongside security improvement, while Pass@10 maintains functional correctness (neutral to +5 points) while achieving substantially larger security improvements.

Practical Implications for Sampling Size Selection

Our findings suggest distinct sampling strategies for different deployment scenarios. For single-attempt or low-latency applications where only one or two samples can be generated, practitioners should expect security improvements of +8 to +10 points but should anticipate functional degradation of -4 to -11 points. In such scenarios, the combined Func-Sec improvement remains modest (+5 to +7 points).

For applications tolerating multiple samples ($k \geq 10$), our approach achieves substantial security improvements (+20 to +33 points) with minimal or positive functional impact (+0 to +5 points). The combined Func-Sec improvements of +15 to +24 points represent a clear net improvement, making higher sampling sizes strongly preferable when computational resources are available.

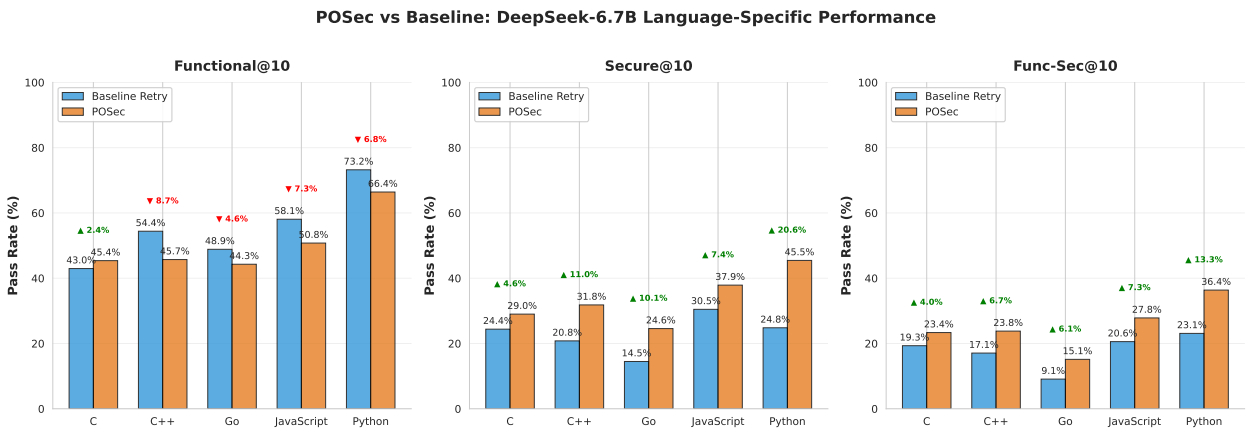


Figure 4.6: Language-specific performance comparison for DeepSeek-Coder-6.7B with MIPROv2 optimizer at Pass@10.

4.5.3 RQ3: How does the effectiveness of POSec vary across programming languages for security-critical code generation?

We analyze how POSec effectiveness varies across the five programming languages in the CWEval benchmark: Python, JavaScript, C++, Go, and C. This analysis is important because programming languages differ fundamentally in their security vulnerability patterns, type systems, memory safety, and error-handling. Understanding language-specific effectiveness enables practitioners to prioritize usage of POSec for languages where security improvements are most substantial and reliable.

Our analysis consists of two stages. First, we conduct detailed language-level performance analysis using DeepSeek-Coder-6.7B with the MIPROv2 optimizer, as this combination achieved the strongest overall results in RQ1 and RQ2. This analysis examines functional correctness trade-offs, security improvements, and combined Func-Sec@10 metrics for each language individually. Second, we evaluate cross-model consistency by measuring Func-Sec@10 improvements across all eight LLMs to determine whether language-level findings generalize beyond a single model configuration. Results are shown in Figures 4.6–4.7.

Language-Specific Performance Analysis

We analyze language-level effectiveness using DeepSeek-Coder-6.7B with the MIPROv2 optimizer at Pass@10, as this combination achieved the strongest overall results. Results are shown in Figure 4.6. POSec improves Secure@10 metrics across all five programming languages, with Python demonstrating the largest improvement. As shown in the middle panel of Figure 4.6, Python achieves +20.64 points in Secure@10 (24.82→45.45), representing an 83.2% relative improvement, alongside the highest combined Func-Sec@10 improvement of +13.27 points (23.10→36.36). Go shows the second-highest relative security improvement at +69.6% (14.49→24.57), while C++ shows substantial absolute improvement of +11.02 points in Secure@10 (20.80→31.82). JavaScript demonstrates moderate security improvement of +7.42 points (30.47→37.89), and C shows the smallest security

improvement at +4.61 points (24.39→29.00).

Functional correctness exhibits language-dependent trade-offs, as illustrated in the left panel of Figure 4.6. C is the only language showing functional improvement with Func@10 increasing +2.42 points (42.96→45.39), while other languages experience modest degradation ranging from -4.58 points (Go) to -8.68 points (C++). Despite these functional trade-offs, the right panel of Figure 4.6 confirms that all languages achieve positive combined Func-Sec@10 improvements, with Python leading at +57.4% relative improvement and Go achieving +66.6% relative improvement from its lower baseline. This pattern suggests that POSec effectively prioritizes security enhancements while maintaining acceptable functional performance across evaluated languages.

Observation 8 : POSec achieves universal security improvements across all five programming languages, with Python showing the largest absolute improvement (+20.64 Secure@10, +13.27 Func-Sec@10) and Go demonstrating the highest relative security improvement (+69.6%), while C uniquely improves both functional correctness and security simultaneously.

The relationship between baseline security performance and improvement magnitude reveals an inverse pattern. Languages with lower baseline Secure@10 scores (Go: 14.49%, C++: 20.80%) achieve higher relative improvements (+69.6% and +53.0% respectively), while languages with higher baseline scores (JavaScript: 30.47%) show more modest relative improvement (+24.3%). This suggests that POSec is particularly effective at addressing security gaps in languages where models initially struggle with secure code generation.

Observation 9 : Security improvement magnitude inversely correlates with baseline performance: languages with lower initial Secure@10 scores achieve higher relative improvement.

Cross-Model Consistency

To assess generalizability beyond a single model, we evaluate Func-Sec@10 improvements across all eight LLMs with the MIPROv2 optimizer. Results are shown in Figure 4.7, where consistency is defined as $\geq 6/8$ models showing positive improvement.

Python demonstrates the most robust cross-model improvements with a mean gain of +7.03% ± 4.42 and all 8/8 models achieving positive results, ranging from +1.79% to +14.15%. As visualized in Figure 4.7, Python's box plot remains entirely above the zero-improvement threshold (dashed line), with individual points from all three model families (DeepSeek, Qwen, Llama) showing positive improvement. JavaScript exhibits similarly consistent behavior with 8/8 models improved, though with lower mean improvement (+3.27% ± 2.59). C++ achieves the second-highest mean improvement (+3.53% ± 4.43) with 6/8 models showing improvement, while C demonstrates modest but consistent improvement (+0.87% ± 2.52) across 6/8 models.

Observation 10 : Python and JavaScript demonstrate perfect cross-model consistency (8/8 models improved), making them the most reliable targets for POSec security optimization regardless of model architecture or family.

Go presents inconsistent results with only 4/8 models showing improvement and significant variance (mean +0.42% ± 2.98, range -3.35% to +6.05%). Figure 4.7 illustrates this inconsistency, with Go's distribution spanning both positive and negative regions and multiple model points falling below the improvement threshold. This inconsistency may stem from Go's stricter type system and distinct error-handling patterns, which could interact unpredictably with security-focused prompt optimization.

Observation 11 : Go exhibits inconsistent cross-model behavior (4/8 models improved) with high variance, suggesting that language-specific characteristics such as strict type systems and explicit error handling may interfere with generalized security prompt optimization techniques.

The cross-model analysis reveals a hierarchy of language reliability for POSec application. Four languages (Python, JavaScript, C++, C) achieve consistent improvements across ≥6/8 models, while Go requires model-specific tuning or alternative optimization approaches. Particularly, model family does not strongly predict language-level success: DeepSeek, Qwen, and Llama models show similar patterns within each language category, as evidenced by the color distribution in Figure 4.7.

Observation 12 : POSec generalizes reliably across model families (DeepSeek, Qwen, Llama) for Python, JavaScript, C++, and C, with language characteristics being a stronger predictor of optimization success than model architecture.

4.5.4 RQ4: What vulnerability and code characteristics distinguish POSec-recoverable tasks from unrecoverable tasks?

Having established POSec's overall effectiveness in RQ1–RQ3, we now investigate *why* certain tasks are recoverable while others are not. Understanding these boundaries is crucial for practitioners seeking to apply POSec in practice. We analyze 119 CWEval tasks using DeepSeek-Coder-6.7B with the BootstrapFewShot optimizer, aggregating results across 10 runs per task.

Task Categorization and Recovery Patterns

We categorize tasks based on baseline and POSec outcomes: **Recovered** (baseline failed, POSec succeeded) and **Unrecoverable** (both failed). POSec recovered 24.4% of all tasks (29/119) and, more notably, 42.0% of baseline failures (29/69).

A striking pattern emerges: recovered tasks exhibit significantly higher baseline functionality (71.0%) compared to unrecoverable tasks (28.5%). This 42.5-point gap suggests that functional code serves as a prerequisite for security recovery. Prompt optimization can teach models to add security

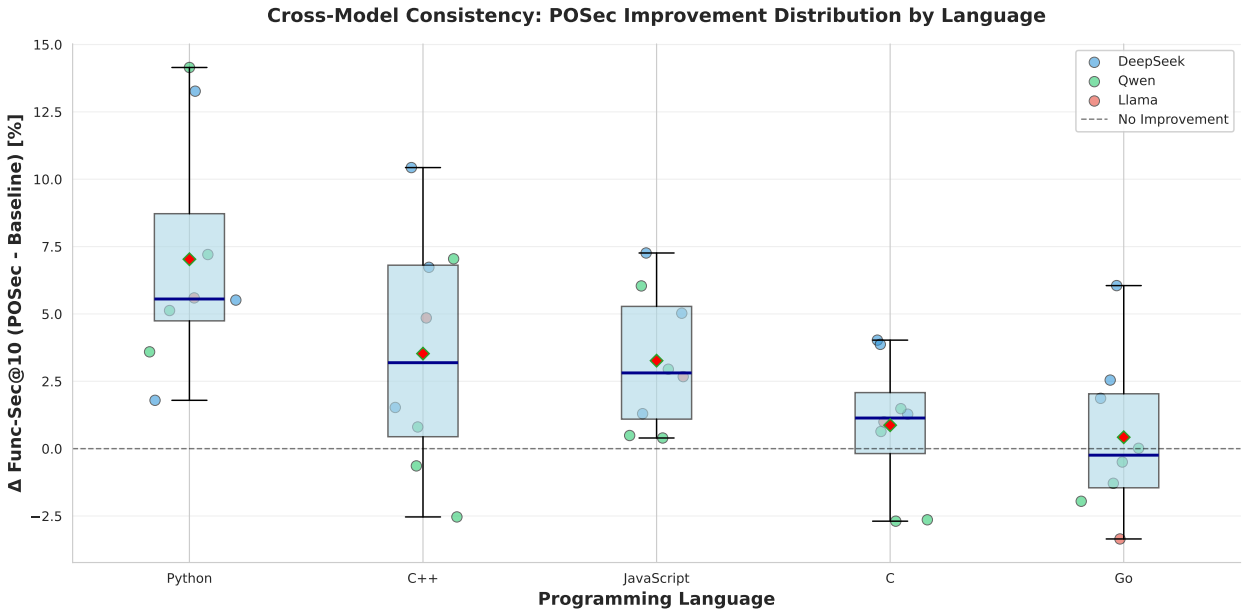


Figure 4.7: Cross-model consistency of Func-Sec@10 improvements across 8 LLMs.

measures, but cannot simultaneously fix fundamental implementation errors.

Beyond baseline functionality, vulnerability type strongly influences recoverability. XSS (CWE-79) achieves 100% recovery rate, followed by CRLF injection (80%) and log injection (60%), while input validation (CWE-20) and weak cryptography (CWE-326) show 0% recovery. This disparity reflects the nature of fixes required: vulnerabilities with well-defined patterns and standard library support (escaping functions, path validation) are recoverable, whereas those requiring abstract reasoning without concrete APIs are not.

Language characteristics further compound these effects. Python achieves the highest recovery rate at 54.5%, followed by JavaScript at 42.9%, C++ at 41.7%, Go at 38.5%, while C achieves only 36.8%. This hierarchy reflects ecosystem differences: high-level languages provide accessible security abstractions (`html.escape()`, `path.resolve()`, `os.path.normpath()`), whereas low-level languages require manual memory management and string manipulation that introduce additional failure modes.

Observation 13: POSec recovery depends on three factors: (1) baseline functionality (recovered: 71.0% vs. unrecoverable: 28.5%), (2) vulnerability specificity (XSS: 100% vs. input validation: 0%), and (3) language abstraction level (Python: 54.5% vs. C: 36.8%). High-level languages with rich standard libraries show higher recovery rates.

Case Study Analysis

To illustrate how these factors interact in practice, we examine two tasks with identical security requirements (path/URL validation to prevent directory escape) but opposite outcomes.

Recovered: CWE-22 Path Traversal (JavaScript). This task requires validating that a user-provided file path does not escape a designated directory. The baseline implementation achieved 90% functionality but 0% security:

```
function read_file_from_dir(filename, dir_path) {
    try {
        const filePath = path.join(dir_path, filename);
        const content = fs.readFileSync(filePath, 'utf8');
        return content;
    } catch (err) { return ''; }
}
```

POSec optimized this to a secure implementation using three security measures: path resolution via `path.resolve()`, containment verification using `startsWith()`, and explicit traversal rejection. The result achieved 60% functionality and 100% security—a 30-point functional decrease representing an acceptable trade-off. This recovery succeeded because the fixes involve well-defined JavaScript APIs.

Unrecoverable: CWE-601 Open Redirect (C). This task requires analogous security logic: validating that URLs belong to an allowed domain. The baseline achieved only 30% functionality and 0% security. While POSec correctly identified the security issue (substring vs. prefix matching), the attempted fix introduced memory management bugs: returning a non-owned pointer and using a static buffer, both causing undefined behavior. Functionality improved marginally (30% $\rightarrow$ 50%), but security remained at 0%.

Synthesis. The contrast is instructive: identical security requirements yielded opposite outcomes. JavaScript's high baseline functionality (90%) and standard library support enabled successful recovery; C's low baseline functionality (30%) and manual memory management led to failure, with the optimization attempt introducing new bugs rather than fixing existing ones.

Observation 14: Low baseline functionality (<40%) shows fundamental task difficulty that prompt optimization cannot overcome. Attempting to add security to weak functional foundations risks introducing new bugs.

4.6 Discussion

4.6.1 Ablation Studies

This section presents ablation studies examining the sensitivity of POSec to key design choices and hyperparameters. We first isolate the contributions of POSec's two core components: prompt optimization and Selective Prompt Anchoring (Section 4.6.1). We then evaluate the impact of

the DSPy module architecture on security improvements (Section 4.6.1). Finally, we analyse the sensitivity of POSec to sampling temperature (Section 4.6.1) and the SPA omega coefficient (Section 4.6.1). All ablation experiments use DeepSeek-Coder-6.7B-Instruct with 10 independent runs per configuration.

Component Contribution Analysis

To quantify the individual contributions of prompt optimization and Selective Prompt Anchoring (SPA), we isolate each component using DeepSeek-Coder-6.7B with MIPROv2. We compare three configurations: baseline (no intervention), DSPy-only (optimized prompts without SPA), and the full POSec pipeline (DSPy + SPA).

Table 4.3: Component contribution analysis at Func-Sec@10 for DeepSeek-Coder-6.7B with MIPROv2.

Configuration	**Func-Sec@10**	**Δ from Baseline**	**Share of Total**
Baseline	41.18%	—	—
DSPy-only	59.73%	+18.55	76.2%
Full POSec (DSPy + SPA)	65.52%	+24.34	100%
SPA marginal contribution		+5.79	23.8%

As shown in Table 4.3, DSPy prompt optimization contributes the majority of the improvement, accounting for 76.2% of the total +24.34 point gain. Adding SPA provides a further +5.79 points (23.8%). This indicates that security-aware prompt restructuring is the primary driver of POSec's effectiveness, while SPA offers meaningful but secondary benefits by maintaining model attention on security specifications during longer generation sequences.

Module Architecture Analysis

To investigate how the DSPy module architecture affects POSec's effectiveness, we compare three modules: *ChainOfThought* (CoT), used in all main experiments; *ReAct*, which interleaves reasoning and acting steps; and *Best-of-N* (BoN), which generates N candidates and selects the best via the evaluation metric. We evaluate all three across the same 8 LLMs, 3 optimizers, and 2 sampling sizes. Table 4.4 reports mean absolute changes (Δ) in Func-Sec@k across all models, along with the number of models showing positive improvement.

Three findings emerge. First, *CoT and BoN are complementary depending on optimizer*: BoN achieves the strongest Func-Sec improvements under BootstrapFewShot (Pass@10: +13.97 vs. CoT's +12.61), while CoT dominates under MIPROv2 (Pass@10: +13.66 vs. BoN's +5.47). This suggests BoN's sampling diversity synergizes with BootstrapFewShot's example-based optimization, whereas MIPROv2's instruction-level optimization benefits more from CoT's structured reasoning.

Second, *ReAct consistently underperforms* for security-focused code generation. Across all configurations, ReAct achieves the lowest mean Func-Sec improvement (ranging from +0.06 to +2.22). While it preserves functional correctness better than CoT and BoN (smallest Func@k drops at

Table 4.4: Module comparison: mean Δ Func-Sec@k across 8 models and consistency (models with positive improvement). Best per row in **bold**.

Optimizer	Pass@k	CoT		ReAct		BoN	
		Mean Δ	#+/8	Mean Δ	#+/8	Mean Δ	#+/8
BootstrapFewShot	@2	+2.60	**7/8**	+0.07	6/8	**+3.15**	**7/8**
	@10	+12.61	**8/8**	+2.00	**8/8**	**+13.97**	**8/8**
MIPROv2	@2	**+2.87**	**8/8**	+0.51	6/8	+0.33	4/6†
	@10	**+13.66**	**8/8**	+2.22	4/8	+5.47	5/6†
GEPA	@2	**+0.22**	**6/8**	+0.06	4/8	−0.19	3/8
	@10	+0.53	4/8	+0.94	5/8	**+1.37**	**7/8**

†BoN+MIPROv2 results unavailable for 2 models due to compute constraints.

Pass@2), it fails to leverage security-enhanced prompts effectively. The iterative reasoning-acting paradigm, effective for multi-step reasoning tasks, does not align well with the single-generation nature of code completion.

Third, *module choice does not alter the core findings.* GEPA remains ineffective regardless of module (<+1.4 mean Func-Sec). The 3–4× scaling from Pass@2 to Pass@10 holds for both CoT and BoN. The functional-security trade-off at Pass@2 and its mitigation at Pass@10 persist across modules, confirming that Observations 1–7 are robust to module selection.

Based on these results, we recommend CoT as the default module due to its highest consistency (7–8/8 models improved across settings) and strong pairing with MIPROv2. When using BootstrapFewShot, BoN is a viable alternative at additional compute cost.

Temperature Analysis

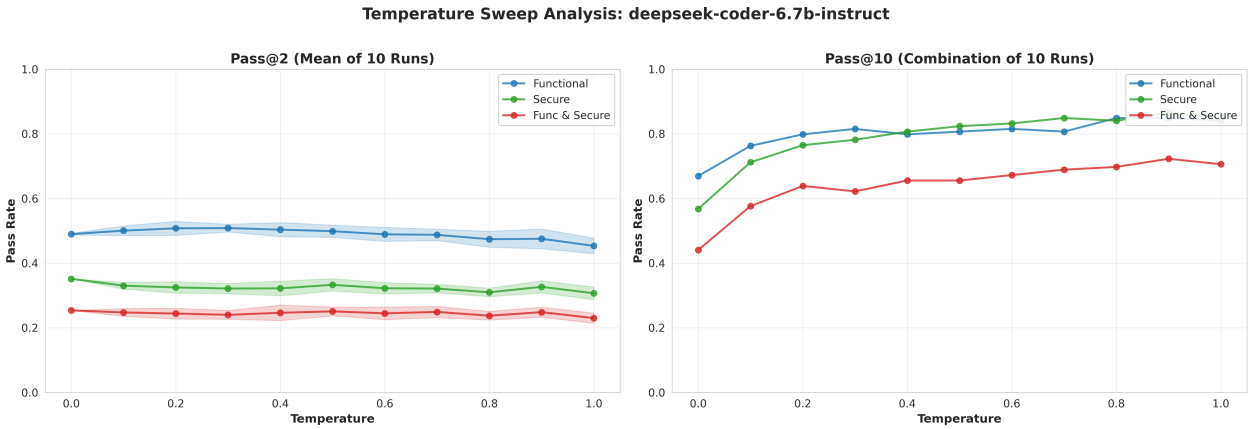


Figure 4.8: Effect of sampling temperature on Pass@2 and Pass@10 performance for DeepSeek-Coder-6.7B-Instruct.

Figure 4.8 presents the effect of sampling temperature on code generation performance across

Pass@2 and Pass@10 metrics. At Pass@2, functional correctness remains relatively stable across temperature values (45%–51%), with a slight peak at $T = 0.2$–0.3 before gradually declining at higher temperatures. Security and combined (Func & Secure) metrics follow similar patterns, showing minimal sensitivity to temperature changes at low sampling budgets.

However, Pass@10 shows a different pattern. All three metrics improve significantly as temperature increases, with functional correctness rising from 66.95% ($T = 0.0$) to 84.87% ($T = 0.8$), secure generation improving from 56.78% to 86.55% ($T = 0.9$), and combined performance increasing from 44.07% to 72.27% ($T = 0.9$). This indicates that sampling at higher temperatures enables the model to discover both functionally correct and secure implementations that deterministic decoding misses. Based on these results, we recommend $T = 0.2$–0.3 for single-attempt scenarios and $T = 0.8$–0.9 for multi-sample settings.

Omega Analysis

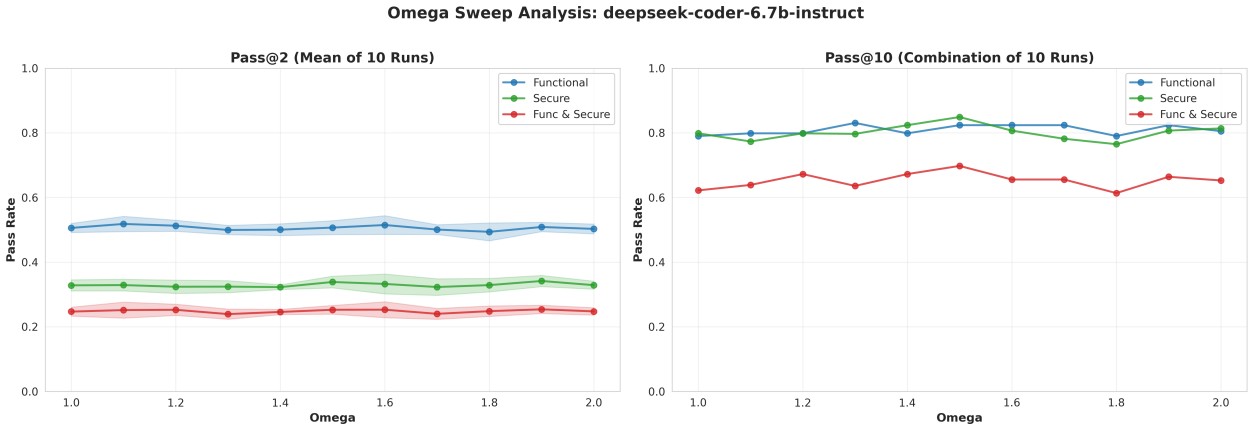


Figure 4.9: Effect of SPA omega coefficient (ω) on Pass@2 and Pass@10 performance for DeepSeek-Coder-6.7B-Instruct.

Figure 4.9 presents the effect of the SPA omega coefficient on generation performance. At Pass@2, all metrics remain stable across omega values ranging from 1.0 to 2.0, with functional correctness varying between 49.93% and 51.80%, and combined performance ranging from 23.91% to 25.38%. This stability suggests that Pass@2 performance is robust to omega selection.

At Pass@10, moderate omega values yield optimal results. Security score peaks at $\omega = 1.5$ (84.87%), and combined functional-secure performance achieves its maximum at the same setting (69.75%). Values below $\omega = 1.4$ show suboptimal security anchoring, while values above $\omega = 1.6$ demonstrate diminishing returns, with $\omega = 1.8$ producing notably lower combined performance (61.34%). This suggests that excessive anchoring strength constrains generation diversity without proportional security benefits. We recommend $\omega = 1.5$ as the optimal setting, providing balanced anchoring strength.

4.6.2 Threats to validity

This section presents the threats to validity for our approach, divided into internal, external, and construct validity threats.

Internal Threats to Validity

Our study faces several internal validity concerns. First, *optimizer training data bias* may exist because we trained optimizers exclusively on successful outputs from the selected models. This approach may introduce architecture-specific biases that affect performance when applied to other model families. Second, due to time and computational constraints, we conducted ablation studies for temperature and omega parameters independently. The *interaction effects* between these hyperparameters and their combined influence on optimal performance remain unexplored. Third, *score variation* presents a concern in our evaluation methodology. While we computed Pass@2 scores as the mean of 10 runs to ensure stability, we did not apply the same averaging approach for Pass@10 results due to resource limitations, potentially leaving variance in these measurements unquantified.

External Threats to Validity

Several factors may limit the generalizability of our findings. Regarding *benchmark representativeness*, we used CWEval as our primary evaluation benchmark. While it contains 119 coding tasks across 31 CWEs, these tasks may not reflect real-world vulnerability distributions and complexity levels encountered in production software. In terms of *language coverage*, we evaluated across only five programming languages, excluding widely used languages such as Rust, Java, and PHP, which limits the generalizability of our results. For *model family diversity*, although we included eight models across three model families with varying parameter sizes, we did not evaluate closed-source models such as Claude and GPT, nor other open-source reasoning models, which may exhibit different response patterns to our approach.

Construct Validity Threats

Test oracle completeness poses a construct validity concern. The security test oracles present in the benchmark may not capture all edge cases, potentially allowing exploitable vulnerabilities to pass the security test cases. This limitation means that passing the benchmark's security tests may not guarantee fully secure code in adversarial conditions.

4.7 Recommendations

Based on our experimental findings, we derive ten practical recommendations to guide the application of POSec for secure code generation. These recommendations, summarized in Table 4.5, are organized into five categories addressing optimizer selection, sampling strategy, model selection, language-specific considerations, and task recovery guidance.

Table 4.5: Recommendations and Supporting Observations, O = Observation

ID	Recommendation	O
Category 1. Optimizer Selection		
R1	We recommend using MIPROv2 or BootstrapFewShot for security-focused prompt optimization, as these instruction-level and example-based techniques consistently achieve 10-30% improvements over evolutionary methods	1, 2
R2	It is advisable to avoid gradient-free evolutionary optimizers (e.g., GEPA) for security-focused code generation, as they tend to provide negligible benefit regardless of model or sampling configuration	1, 2
R3	When choosing between MIPROv2 and BootstrapFewShot, practitioners may consider that MIPROv2 tends to achieve slightly higher performance while BootstrapFewShot provides more consistent results across models	1, 3
Category 2. Sampling Strategy		
R4	Consider allocating extended sampling ($k \geq 10$) when computational resources permit, as security improvements tend to scale approximately 3-4× from Pass@2 to Pass@10	4, 5, 7
R5	Practitioners may need to consider functional-security trade-offs at low sampling sizes (Pass@2: -4 to -11 functional points); these trade-offs tend to be effectively mitigated at Pass@10	4, 7
Category 3. Model Selection		
R6	Consider selecting larger model variants (≥6.7B parameters) within model families when security improvement is the primary objective, as model capacity positively influence POSec effectiveness	6
R7	For models with already high baseline security scores, practitioners should expect diminishing returns from POSec due to ceiling effects; focusing optimization efforts on security-weak models may yield better results	6, 9
Category 4. Language-Specific Guidance		
R8	We suggest prioritizing Python and JavaScript as primary targets for POSec deployment, as they demonstrate perfect cross-model consistency (8/8 models improved) and highest reliability in our experiments	10, 12
R9	Applying POSec to languages with lower baseline security scores (Go, C++) may yield higher relative improvements and could be worth considering for security-critical applications	8, 9
R10	It is advisable to exercise caution when applying POSec to Go and to implement model-specific validation, given the inconsistent cross-model behavior observed (4/8 models improved)	11
Category 5. Recovery-Based Guidance		
R11	Apply POSec selectively to tasks with ≥70% baseline functionality; tasks below 40% may regress due to introduced bugs and require alternative approaches.	13, 14
R12	We suggest prioritizing well-defined vulnerability patterns (XSS, path traversal) in high-level languages with rich standard libraries (Python, JavaScript). For lower-recovery languages (C: 36.8%, Go: 38.5%), combine POSec with static analysis or constrained decoding.	

Optimizer Selection (R1, R2, R3). When implementing POSec, practitioners could face a critical decision regarding optimizer selection. We recommend using MIPROv2 or BootstrapFewShot for security-focused prompt optimization, as these instruction-level and example-based techniques consistently achieve 10-30% improvements over evolutionary methods (O1, O2). It is advisable to avoid gradient-free evolutionary optimizers such as GEPA, which tend to provide negligible benefit regardless of model or sampling configuration (O1, O2). When choosing between the two recommended optimizers, practitioners may consider that MIPROv2 tends to achieve slightly higher performance while BootstrapFewShot provides more consistent results across different model families (O1, O3).

Sampling Strategy (R4, R5). Sampling budget significantly influences POSec outcomes. We recommend allocating higher sampling budgets ($k \geq 10$) when computational resources permit, as security improvements tend to scale approximately 3-4× from Pass@2 to Pass@10 (O4, O5, O7). However, practitioners may need to accept modest functional-security trade-offs at low sampling sizes, with Pass@2 showing -4 to -11 functional points degradation. These trade-offs tend to be effectively mitigated at Pass@10, where both security and functional correctness can improve simultaneously (O4, O7).

Model Selection (R6, R7). Model selection substantially impacts POSec effectiveness. We suggest selecting larger model variants (≥6.7B parameters) within model families when security improvement is the primary objective, as model capacity positively influences POSec effectiveness (O6). Additionally, for models with already high baseline security scores, practitioners should expect diminishing returns from POSec due to ceiling effects. Focusing optimization efforts on security-weak models may yield better results, as languages and models with lower baseline scores achieve higher relative improvements (O6, O9).

Language-Specific Guidance (R8, R9, R10). Language characteristics significantly influence POSec effectiveness, often more so than model architecture. We suggest prioritizing Python and JavaScript as primary targets for POSec deployment, as they demonstrate perfect cross-model consistency (8/8 models improved) and highest reliability in our experiments (O10, O12). Applying POSec to languages with lower baseline security scores, such as Go and C++, may yield higher relative improvements and could be worth considering for security-critical applications (O8, O9). However, it is advisable to exercise caution when applying POSec to Go and to implement model-specific validation, given the inconsistent cross-model behavior observed where only 4 out of 8 models showed improvement (O11).

Recovery-Based Guidance (R11, R12). Our case study analysis reveals that POSec recovery success depends on identifiable task characteristics, enabling targeted application. We recommend applying POSec selectively to tasks with baseline functionality ≥70%; tasks below 40% functionality are unlikely to benefit and may even regress due to introduced bugs, such as memory management errors observed in C (O13, O14). For low-functionality tasks, practitioners should consider alternative approaches such as multi-stage pipelines that first repair functionality before applying security optimization.

Additionally, we recommend prioritizing well-defined vulnerability patterns with concrete mitigation

APIs: XSS (100% recovery), CRLF injection (80%), and path traversal (44%) over abstract security requirements such as input validation (0%) and cryptography (0%) that lack learnable patterns (O13). Language selection significantly impacts recovery: high-level languages with rich standard libraries achieve substantially higher rates (Python: 54.5%, JavaScript: 42.9%) compared to low-level languages requiring manual memory management (C: 36.8%, C++: 41.7%). For security-critical C/C++ applications, combining POSec with post-generation static analysis or constrained decoding is advisable given their systematically lower recovery rates (O14).

4.8 Conclusion

We presented POSec, a modular pipeline framework for secure code generation that integrates into existing workflows without model retraining. The framework consists of an Optimizer Training Pipeline (Modeling and Tuning components) that learns security-aware prompting strategies, and an Inference Pipeline that applies trained optimizers with Selective Prompt Anchoring to maintain attention on security specifications. Our evaluation across 8 LLMs, 5 languages, and 3 optimizers demonstrates that MIPROv2 and BootstrapFewShot achieve +20–33% security improvements at Pass@10, with effectiveness varying by language (Python and JavaScript showing perfect cross-model consistency) and task characteristics (high baseline functionality and well-defined vulnerability patterns). POSec offers practitioners a practical approach to improving code security without fine-tuning, bridging the gap between functional correctness and security in LLM-generated code.

Chapter 5

Conclusions and Future work

This thesis investigated how the quality gap between functionally correct and production-ready LLM-generated code can be systematically reduced through complementary interventions at different levels of the generation pipeline. Two studies were conducted: one on training-time alignment (Paper 1) and one on inference-time prompt optimization (Paper 2).

5.1 Summary of Findings

Training-Time Alignment (Paper 1). Alignment from finetuned models improved performance in 51% of cases for functional tasks and 53% for non-functional tasks, with notably different magnitudes: modest (4.9%) for functional tasks and more pronounced (10.6%) for non-functional tasks. Pretrained-to-aligned pathways achieved larger relative gains (CodeLlama-7b: +75% non-functional, Llama3-8b: +42% functional) but from weaker baselines. Non-functional alignment was consistent across all model families, while functional alignment ranged from significant success to catastrophic failure (CodeLlama-7b: −40%). The study also revealed model family-specific technique preferences (Meta-Llama with DPO, DeepSeek with BoNBoN) and early failure indicators where SFT-stage degradation predicted final alignment failure.

Inference-Time Prompt Optimization (Paper 2). MIPROv2 and BootstrapFewShot achieved security improvements of +8 to 10% at Pass@2 and +20 to 33% at Pass@10, while GEPA provided negligible benefit. Security improvements increased approximately 3 to 4× with increased sampling size. Python and JavaScript demonstrated perfect cross-model consistency (8/8 models improved), while Go showed inconsistent behaviour (4/8 models). Recovery analysis revealed that POSec succeeds on tasks with high baseline functionality (69.1%) and well-defined vulnerability patterns (XSS: 100% recovery) but fails on abstract security requirements (input validation: 0%).

Overall Findings. Examining both studies together reveals four convergent themes:

- **Non-functional improvement is more achievable than functional improvement** at both intervention levels: Paper 1 showed consistent non-functional gains (10.6%) versus un-

reliable functional gains (4.9%), while Paper 2 improved security across 8 models through prompt optimization.

- **Baseline capability is a prerequisite**, as both studies found that models must produce functionally correct code before quality improvements can succeed.
- **Model family matters more than technique**: no single alignment method or optimizer dominated across all models and languages, and technique selection should be guided by specific model and task characteristics.
- **Trade-offs are inevitable but manageable**, with increased compute and time resources (stronger baselines in Paper 1, larger sampling budgets in Paper 2) reducing the respective stability-plasticity and functional-security trade-offs.

Training-time alignment offers breadth across five non-functional dimensions with modest per-dimension gains. Inference-time optimization offers depth with substantial gains (+20 to 33%) on targeted properties like security. A comprehensive strategy may involve both: alignment to establish a broadly higher quality baseline, followed by inference-time optimization for specific critical requirements.

5.2 Limitations

Several limitations apply across both studies in this thesis.

Model scale and diversity. Both studies evaluated models in the 1.3B to 33B parameter range. Larger models exceeding 70B parameters may exhibit emergent properties [79] that alter alignment responsiveness or prompt optimization effectiveness. Furthermore, neither study evaluated closed-source models such as GPT-4 or Claude, which represent a substantial share of production code generation usage. The generalizability of our findings to these models remains an open question.

Benchmark representativeness. The training-time study relied on the CODAL benchmark for non-functional evaluation, whose five dimensions are derived from established software engineering practices but were not directly validated through developer elicitation. The inference-time study used CWEval, which contains 119 tasks across 31 CWEs but may not reflect the full distribution and complexity of vulnerabilities encountered in production software. Both benchmarks serve as proxies for code quality rather than comprehensive measures of real-world production readiness.

Language coverage. The inference-time study evaluated five programming languages, excluding widely used languages such as Rust, Java, and PHP. The training-time study focused exclusively on Python-centric benchmarks. Security vulnerability patterns and non-functional quality characteristics differ across languages, and our findings may not transfer to languages not evaluated.

Interaction between interventions. The two studies were conducted independently. Whether training-time alignment gains and inference-time optimization gains are additive, redundant, or interact in unexpected ways was not evaluated. The optimal sequencing and combination of these interventions remains unknown.

Evaluation methodology. The training-time study used LLM-as-judge evaluation for non-functional quality, which, despite validation against human annotators, introduces potential biases in scoring. The inference-time study used a heuristic-based metric during optimizer training rather than executing code in sandboxed environments, which may introduce gaps between training-time proxy signals and actual security outcomes.

Preference data and generalization. All preference datasets in the training-time study were model-specific and self-generated following the SelfCodeAlign pipeline. While this maximizes within-model alignment effectiveness, it may encode model-specific stylistic biases rather than broadly generalizable quality patterns. Cross-model preference transfer was not explored.

5.3 Future Work

Several directions for future work emerge directly from the limitations identified above.

Stacking training-time and inference-time interventions. The most immediate question is whether the gains from alignment and prompt optimization are additive. Applying POSec to an already-aligned model would reveal whether the two levels provide complementary benefits or exhibit diminishing returns, directly addressing the limitation that our two studies were conducted independently.

Scaling to larger and proprietary models. Extending the evaluation to models exceeding 70B parameters, as well as closed-source models accessible only via API, would clarify whether the patterns observed in our studies, such as model family-specific technique preferences and the baseline functionality prerequisite, generalize to the systems most widely deployed in production.

Broader language and benchmark coverage. Evaluating both approaches on additional programming languages (Rust, Java, PHP) and on more comprehensive benchmarks that capture real-world vulnerability distributions and organizational coding standards would strengthen the external validity of the findings.

Expanding inference-time optimization beyond security. POSec was evaluated exclusively for security. Extending the prompt optimization framework to target other non-functional dimensions such as readability, maintainability, and efficiency would provide a lightweight alternative

to alignment for practitioners without model access, and would address the limitation that our inference-time study focused on a single quality property.

Developer-validated quality benchmarks. Both studies relied on benchmark-defined proxies for code quality. Developing evaluation frameworks grounded in developer elicitation, where practitioners define and weight the non-functional properties most relevant to their organizational context, would improve construct validity and practical relevance.

Automated intervention selection. Finally, building systems that automatically select the appropriate intervention (alignment, prompt optimization, or their combination) based on task characteristics, model capabilities, and resource constraints would operationalize the decision framework proposed in this thesis and make the findings directly actionable for practitioners.

Bibliography

[1] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, *et al.*, "Gpt-4 technical report," *arXiv preprint arXiv:2303.08774*, 2023.

[2] A. Anthropic, "The claude 3 model family: Opus, sonnet, haiku," *Claude-3 Model Card*, vol. 1, no. 1, p. 4, 2024.

[3] G. Comanici, E. Bieber, M. Schaekermann, I. Pasupat, N. Sachdeva, I. Dhillon, M. Blistein, O. Ram, D. Zhang, E. Rosen, *et al.*, "Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities," *arXiv preprint arXiv:2507.06261*, 2025.

[4] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," 2024.

[5] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li, *et al.*, "Deepseek-coder: When the large language model meets programming–the rise of code intelligence," *arXiv preprint arXiv:2401.14196*, 2024.

[6] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi, "Codet5+: Open code large language models for code understanding and generation," *arXiv preprint arXiv:2305.07922*, 2023.

[7] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "Wizardcoder: Empowering code large language models with evol-instruct," *arXiv preprint arXiv:2306.08568*, 2023.

[8] N. Muennighoff, Q. Liu, A. Zebaze, Q. Zheng, B. Hui, T. Y. Zhuo, S. Singh, X. Tang, L. Von Werra, and S. Longpre, "Octopack: Instruction tuning code large language models," in *NeurIPS 2023 workshop on instruction tuning and instruction following*, 2023.

[9] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Empowering code generation with oss-instruct," *arXiv preprint arXiv:2312.02120*, 2023.

[10] "AI — 2025 Stack Overflow Developer Survey — survey.stackoverflow.co." https://survey.stackoverflow.co/2025/ai/. [Accessed 15-01-2026].

[11] Y. Wei, F. Cassano, J. Liu, Y. Ding, N. Jain, Z. Mueller, H. de Vries, L. V. Werra, A. Guha, and L. ZHANG, "Selfcodealign: Self-alignment for code generation," in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[12] Z. Ma, H. Guo, J. Chen, G. Peng, Z. Cao, Y. Ma, and Y.-J. Gong, "Llamoco: Instruction tuning of large language models for optimization code generation," *arXiv preprint arXiv:2403.01131*, 2024.

[13] J. T. Almonte, S. A. Boominathan, and N. Nascimento, "Automated non-functional requirements generation in software engineering with large language models: A comparative study," *arXiv preprint arXiv:2503.15248*, 2025.

[14] J. Ullrich, M. Koch, and A. Vogelsang, "From requirements to code: Understanding developer practices in llm-assisted software engineering," *arXiv preprint arXiv:2507.07548*, 2025.

[15] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," *CoRR*, vol. abs/2107.03374, 2021.

[16] Z. Yang, Z. Sun, T. Z. Yue, P. Devanbu, and D. Lo, "Robustness, security, privacy, explainability, efficiency, and usability of large language models for code," 2024.

[17] J. Austin, A. Odena, M. I. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. J. Cai, M. Terry, Q. V. Le, and C. Sutton, "Program synthesis with large language models," *CoRR*, vol. abs/2108.07732, 2021.

[18] C. S. Xia, Y. Deng, and L. ZHANG, "Top leaderboard ranking = top coding proficiency, always? evoeval: Evolving coding benchmarks via LLM," in *First Conference on Language Modeling*, 2024.

[19] M. Weyssow, A. Kamanda, X. Zhou, and H. Sahraoui, "Codeultrafeedback: An llm-as-a-judge dataset for aligning large language models to coding preferences," *ACM Trans. Softw. Eng. Methodol.*, May 2025. Just Accepted.

[20] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," *Communications of the ACM*, vol. 68, no. 2, pp. 96–105, 2025.

[21] S.-C. Dai, J. Xu, and G. Tao, "A comprehensive study of llm secure code generation," *arXiv preprint arXiv:2503.15554*, 2025.

[22] R. Rafailov, A. Sharma, E. Mitchell, C. D. Manning, S. Ermon, and C. Finn, "Direct preference optimization: Your language model is secretly a reward model," in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[23] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, "Deep reinforcement learning from human preferences," *Advances in neural information processing systems*, vol. 30, 2017.

[24] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, *et al.*, "Training language models to follow instructions with human feedback," *Advances in neural information processing systems*, vol. 35, pp. 27730–27744, 2022.

[25] Z. Wang, B. Bi, S. K. Pentyala, K. Ramnath, S. Chaudhuri, S. Mehrotra, X.-B. Mao, S. Asur, *et al.*, "A comprehensive survey of llm alignment techniques: Rlhf, rlaif, ppo, dpo and more," *arXiv preprint arXiv:2407.16216*, 2024.

[26] Y. Zhang, Z. Chen, Y. Fang, Y. Lu, L. Fangming, W. Zhang, and H. Chen, "Knowledgeable preference alignment for LLMs in domain-specific question answering," in *Findings of the Association for Computational Linguistics: ACL 2024* (L.-W. Ku, A. Martins, and V. Srikumar, eds.), (Bangkok, Thailand), pp. 891–904, Association for Computational Linguistics, Aug. 2024.

[27] Y. Tian and T. Zhang, "Selective prompt anchoring for code generation," 2025.

[28] "GitHub Copilot — github.com." https://github.com/copilot. [Accessed 13-01-2026].

[29] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu, *et al.*, "Qwen2. 5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.

[30] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[31] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, *et al.*, "Starcoder: may the source be with you!," *arXiv preprint arXiv:2305.06161*, 2023.

[32] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan, *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

[33] S. Zhang, P. Sun, S. Chen, M. Xiao, W. Shao, W. Zhang, Y. Liu, K. Chen, and P. Luo, "Gpt4roi: Instruction tuning large language model on region-of-interest," in *European conference on computer vision*, pp. 52–70, Springer, 2024.

[34] W. U. Ahmad, A. Ficek, M. Samadi, J. Huang, V. Noroozi, S. Majumdar, and B. Ginsburg, "Opencodeinstruct: A large-scale instruction tuning dataset for code llms," *arXiv preprint arXiv:2504.04030*, 2025.

[35] K. Li, Q. Hu, X. Zhao, H. Chen, Y. Xie, T. Liu, Q. Xie, and J. He, "Instructcoder: Instruction tuning large language models for code editing," *arXiv preprint arXiv:2310.20329*, 2023.

[36] J. Liu, C. S. Xia, Y. Wang, and L. ZHANG, "Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation," in *Advances in Neural Information Processing Systems* (A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, eds.), vol. 36, pp. 21558–21572, Curran Associates, Inc., 2023.

[37] J. Liu, S. Xie, J. Wang, Y. Wei, Y. Ding, and L. Zhang, "Evaluating language models for efficient code generation," *arXiv preprint arXiv:2408.06450*, 2024.

[38] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou, "Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation," *arXiv preprint arXiv:2308.01861*, 2023.

[39] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, Q. Wang, and T. Xie, "Codereval: A benchmark of pragmatic code generation with generative pre-trained models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pp. 1–12, 2024.

[40] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?," *arXiv preprint arXiv:2310.06770*, 2023.

[41] M. Singhal, T. Aggarwal, A. Awasthi, N. Natarajan, and A. Kanade, "Nofuneval: Funny how code lms falter on requirements beyond functional correctness," *arXiv preprint arXiv:2401.15963*, 2024.

[42] J. Peng, L. Cui, K. Huang, J. Yang, and B. Ray, "Cweval: Outcome-driven evaluation on functionality and security of llm code generation," 2025.

[43] D. Kocetkov, R. Li, L. B. Allal, J. Li, C. Mou, C. M. Ferrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf, D. Bahdanau, L. von Werra, and H. de Vries, "The stack: 3 tb of permissively licensed source code," 2022.

[44] H. Lee, S. Phatale, H. Mansoor, T. Mesnard, J. Ferret, K. Lu, C. Bishop, E. Hall, V. Carbune, A. Rastogi, *et al.*, "Rlaif vs. rlhf: Scaling reinforcement learning from human feedback with ai feedback," *arXiv preprint arXiv:2309.00267*, 2023.

[45] K. Ethayarajh, W. Xu, N. Muennighoff, D. Jurafsky, and D. Kiela, "Kto: Model alignment as prospect theoretic optimization," *arXiv preprint arXiv:2402.01306*, 2024.

[46] J. Hong, N. Lee, and J. Thorne, "Orpo: Monolithic preference optimization without reference model," *arXiv preprint arXiv:2403.07691*, 2024.

[47] Y. Yin, Z. Wang, Y. Gu, H. Huang, W. Chen, and M. Zhou, "Relative preference optimization: Enhancing llm alignment through contrasting responses across identical and diverse prompts," *arXiv preprint arXiv:2402.10958*, 2024.

[48] S. Xu, W. Fu, J. Gao, W. Ye, W. Liu, Z. Mei, G. Wang, C. Yu, and Y. Wu, "Is dpo superior to ppo for llm alignment? a comprehensive study," *arXiv preprint arXiv:2404.10719*, 2024.

[49] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017.

[50] N. Stiennon, L. Ouyang, J. Wu, D. Ziegler, R. Lowe, C. Voss, A. Radford, D. Amodei, and P. F. Christiano, "Learning to summarize with human feedback," *Advances in neural information processing systems*, vol. 33, pp. 3008–3021, 2020.

[51] L. Gui, C. Garbacea, and V. Veitch, "BoNBon alignment for large language models and the sweetness of best-of-n sampling," in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[52] R. Y. Pang, W. Yuan, H. He, K. Cho, S. Sukhbaatar, and J. Weston, "Iterative reasoning preference optimization," *Advances in Neural Information Processing Systems*, vol. 37, pp. 116617–116637, 2024.

[53] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, *et al.*, "Overcoming catastrophic forgetting in neural networks," *Proceedings of the national academy of sciences*, vol. 114, no. 13, pp. 3521–3526, 2017.

[54] V. V. Ramasesh, A. Lewkowycz, and E. Dyer, "Effect of scale on catastrophic forgetting in neural networks," in *International conference on learning representations*, 2021.

[55] Y. Luo, Z. Yang, F. Meng, Y. Li, J. Zhou, and Y. Zhang, "An empirical study of catastrophic forgetting in large language models during continual fine-tuning," 2025.

[56] S. Kotha, J. M. Springer, and A. Raghunathan, "Understanding catastrophic forgetting in language models via implicit inference," *arXiv preprint arXiv:2309.10105*, 2023.

[57] I. Jindal, C. Badrinath, P. Bharti, L. Vinay, and S. D. Sharma, "Balancing continuous pretraining and instruction fine-tuning: Optimizing instruction-following in llms," *arXiv preprint arXiv:2410.10739*, 2024.

[58] J. He, M. Vero, G. Krasnopolska, and M. Vechev, "Instruction tuning for secure code generation," *arXiv preprint arXiv:2402.09497*, 2024.

[59] J. He and M. Vechev, "Large language models for code: Security hardening and adversarial testing," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1865–1879, 2023.

[60] K. Chang, S. Xu, C. Wang, Y. Luo, X. Liu, T. Xiao, and J. Zhu, "Efficient prompting methods for large language models: A survey," *arXiv preprint arXiv:2404.01077*, 2024.

[61] K. Ramnath, K. Zhou, S. Guan, S. S. Mishra, X. Qi, Z. Shen, S. Wang, S. Woo, S. Jeoung, Y. Wang, *et al.*, "A systematic survey of automatic prompt optimization techniques," *arXiv preprint arXiv:2502.16923*, 2025.

[62] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in neural information processing systems*, vol. 35, pp. 24824–24837, 2022.

[63] C. Tony, N. E. Díaz Ferreyra, M. Mutas, S. Dhif, and R. Scandariato, "Prompting techniques for secure code generation: A systematic investigation," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 8, pp. 1–53, 2025.

[64] P. Bareiß, B. Souza, M. d'Amorim, and M. Pradel, "Code generation tools (almost) for free? a study of few-shot, pre-trained language models on code," *arXiv preprint arXiv:2206.01335*, 2022.

[65] O. Khattab, A. Singhvi, P. Maheshwari, Z. Zhang, K. Santhanam, S. Vardhamanan, S. Haq, A. Sharma, T. T. Joshi, H. Moazam, H. Miller, M. Zaharia, and C. Potts, "Dspy: Compiling declarative language model calls into self-improving pipelines," 2023.

[66] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," 2020.

[67] K. Opsahl-Ong, M. J. Ryan, J. Purtell, D. Broman, C. Potts, M. Zaharia, and O. Khattab, "Optimizing instructions and demonstrations for multi-stage language model programs," *arXiv preprint arXiv:2406.11695*, 2024.

[68] L. A. Agrawal, S. Tan, D. Soylu, N. Ziems, R. Khare, K. Opsahl-Ong, A. Singhvi, H. Shandilya, M. J. Ryan, M. Jiang, C. Potts, K. Sen, A. G. Dimakis, I. Stoica, D. Klein, M. Zaharia, and O. Khattab, "Gepa: Reflective prompt evolution can outperform reinforcement learning," 2025.

[69] D. Li, M. Yan, Y. Zhang, Z. Liu, C. Liu, X. Zhang, T. Chen, and D. Lo, "Cosec: On-the-fly security hardening of code llms via supervised co-decoding," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1428–1439, 2024.

[70] Y. Fu, E. Baker, Y. Ding, and Y. Chen, "Constrained decoding for secure code generation," *arXiv preprint arXiv:2405.00218*, 2024.

[71] M. Nazzal, I. Khalil, A. Khreishah, and N. Phan, "Promsec: Prompt optimization for secure generation of functional source code with large language models (llms)," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pp. 2266–2280, 2024.

[72] X. Deng, H. Zhong, R. Ai, F. Feng, Z. Wang, and X. He, "Less is more: Improving llm alignment via preference data selection," *arXiv preprint arXiv:2502.14560*, 2025.

[73] Y. Miao, B. Gao, S. Quan, J. Lin, D. Zan, J. Liu, J. Yang, T. Liu, and Z. Deng, "Aligning codellms with direct preference optimization," *arXiv preprint arXiv:2410.18585*, 2024.

[74] K. Zhang, G. Li, Y. Dong, J. Xu, J. Zhang, J. Su, Y. Liu, and Z. Jin, "Codedpo: Aligning code models with self generated and verified source code," *arXiv preprint arXiv:2410.05605*, 2024.

[75] Y.-D. Tsai, M. Liu, and H. Ren, "Code less, align more: Efficient llm fine-tuning for code generation with data pruning," *arXiv preprint arXiv:2407.05040*, 2024.

[76] L. Gondara, J. Simkin, G. Sayle, S. Devji, G. Arbour, and R. Ng, "Small or large? zero-shot or finetuned? guiding language model choice for specialized applications in healthcare," *arXiv preprint arXiv:2504.21191*, 2025.

[77] J. M. Springer, S. Goyal, K. Wen, T. Kumar, X. Yue, S. Malladi, G. Neubig, and A. Raghunathan, "Overtrained language models are harder to fine-tune," *arXiv preprint arXiv:2503.19206*, 2025.

[78] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, *et al.*, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," *arXiv preprint arXiv:2501.12948*, 2025.

[79] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, *et al.*, "Emergent abilities of large language models," *arXiv preprint arXiv:2206.07682*, 2022.

[80] S. Muckatira, V. Deshpande, V. Lialin, and A. Rumshisky, "Emergent abilities in reduced-scale generative language models," *arXiv preprint arXiv:2404.02204*, 2024.

[81] W. Ren, X. Li, L. Wang, T. Zhao, and W. Qin, "Analyzing and reducing catastrophic forgetting in parameter efficient tuning," 2024.

[82] L. A. Goodman, "Snowball sampling," *The Annals of Mathematical Statistics*, vol. 32, no. 1, pp. 148–170, 1961.

[83] Y. Dubois, B. Galambosi, P. Liang, and T. B. Hashimoto, "Length-controlled alpacaeval: A simple way to debias automatic evaluators," *arXiv preprint arXiv:2404.04475*, 2024.

[84] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing, *et al.*, "Judging llm-as-a-judge with mt-bench and chatbot arena," *Advances in neural information processing systems*, vol. 36, pp. 46595–46623, 2023.

[85] J. Cohen, "Weighted kappa: Nominal scale agreement provision for scaled disagreement or partial credit.," *Psychological bulletin*, vol. 70, no. 4, p. 213, 1968.

[86] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012.

[87] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "Codegen: An open large language model for code with multi-turn program synthesis," *arXiv preprint arXiv:2203.13474*, 2022.

[88] J. Li, Y. Zhao, Y. Li, G. Li, and Z. Jin, "Acecoder: An effective prompting technique specialized in code generation," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–26, 2024.

[89] J. Li, G. Li, Y. Li, and Z. Jin, "Structured chain-of-thought prompting for code generation," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 2, pp. 1–23, 2025.

[90] X. Jiang, Y. Dong, L. Wang, Z. Fang, Q. Shang, G. Li, Z. Jin, and W. Jiao, "Self-planning code generation with large language models," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 7, pp. 1–30, 2024.

[91] A. Datta, A. Aljohani, and H. Do, "Secure code generation at scale with reflexion," *arXiv preprint arXiv:2511.03898*, 2025.