KoCo-Bench: Can Large Language Models Leverage Domain Knowledge in Software Development?

Xue Jiang[1], Ge Li[1], Jiaru Qian[1], Xianjie Shi[1], Chenjie Li[1], Hao Zhu[1], Ziyu Wang[1], Jielun Zhang[1], Zheyu Zhao[1], Kechi Zhang[1], Jia Li[2], Wenpin Jiao[1], Zhi Jin[1], Yihong Dong[1]

[1]Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education; School of Computer Science, Peking University, Beijing, China
[2]School of Computer Science, Wuhan University

Abstract

LLMs excel at general programming but struggle with domain-specific software development, necessitating domain specialization methods for LLMs to learn and utilize domain knowledge and data. However, *existing domain-specific code benchmarks cannot evaluate the effectiveness of domain specialization methods*, which focus on assessing what knowledge LLMs possess rather than how they acquire and apply new knowledge, lacking explicit knowledge corpora for developing domain specialization methods. To this end, we present KoCo-Bench, a novel benchmark designed for evaluating domain specialization methods in real-world software development. KoCo-Bench contains 6 emerging domains with 11 software frameworks and 25 projects, featuring curated knowledge corpora alongside multi-granularity evaluation tasks including domain code generation (from function-level to project-level with rigorous test suites) and domain knowledge understanding (via multiple-choice Q&A). Unlike previous benchmarks that only provide test sets for direct evaluation, KoCo-Bench requires acquiring and applying diverse domain knowledge (APIs, rules, constraints, *etc*.) from knowledge corpora to solve evaluation tasks. Our evaluations reveal that KoCo-Bench poses significant challenges to state-of-the-art LLMs. Even with domain specialization methods (*e.g*., SFT, RAG, kNN-LM) applied, improvements remain marginal. Best-performing coding agent, Claude Code, achieves only 34.2%, highlighting the urgent need for more effective domain specialization methods. We release KoCo-Bench, evaluation code, and baselines to advance further research at https://github.com/jiangxxxue/KOCO-bench.

1 Introduction

As Large Language Models (LLMs) demonstrate remarkable capabilities in code generation (Claude-Code; GPT-5; Gemini3), the research focus of LLM for Software Engineering (LLM4SE) is shifting from general programming toward domain-specific development (Gu et al.; Yu et al., 2025). Unlike general programming tasks, domain-specific development demands specialized knowledge, including proprietary rules, processes, API, protocols, dependencies, constraints, etc. The domain knowledge exists in various forms scattered across documentation, source code, or example code. Therefore, how effectively LLMs can acquire, comprehend, and apply domain knowledge is critical to their real-world utility in software engineering.

LLMs exhibit substantial limitations when processing domain knowledge (Gu et al.; Hung et al., 2023), attributable to several factors. First, domain knowledge is inherently specialized, with limited training corpora available, rendering it difficult for models to develop adequate comprehension capabilities through pre-training. Second, the organizational complexity of codebases and data, coupled with fragmented knowledge and implicit semantics, renders models susceptible to incomplete or erroneous understanding during both learning and inference. Moreover, given the rapid evolution of software ecosystems, the inability of LLMs to efficiently acquire and adapt to emerging domain knowledge significantly impairs their effectiveness in practical development contexts.

These limitations have motivated research into domain specialization methods aimed at enabling LLMs to learn and utilize domain knowledge and data. Existing domain specialization methods pre-

Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 28422–28441

dominantly rely on general-purpose techniques such as Supervised Fine-Tuning (SFT) (Dong et al., 2024a; Pareja et al., 2024), Retrieval-Augmented Generation (RAG) (Lewis et al., 2020; Gao et al., 2023), and various extensions thereof (Hu et al., 2021; Khandelwal et al., 2020). These approaches provide insufficient gains and exhibit notable drawbacks in domain-specific software development. For instance, SFT is constrained by the high cost and scarcity of high-quality domain-labeled data, often leading to overfitting or shallow pattern learning (Ghosh et al., 2024; Lin et al., 2025). RAG improves factual recall but struggles with complex reasoning over fragmented and implicit domain knowledge embedded in large codebases (Zhang et al., 2025; Agrawal et al., 2024). This underscores the urgent need for domain specialization methods that are not only more effective but also tailored to software engineering.

A key obstacle to progress is the lack of code benchmarks for evaluating domain specialization methods. Existing domain-specific code benchmarks are primarily constructed to assess what domain knowledge LLMs already know, rather than how LLMs can better learn and adapt to new domain knowledge. Representative benchmarks, *e.g.*, EvoCodeBench (Li et al., 2024) and DomainEval (Zhu et al., 2024), follow a construction pipeline that selects projects from target domains, extracts functions along with corresponding unit tests, and formulates them into code generation tasks with annotated requirements. As a result, these benchmarks consist solely of evaluation test sets and associated code contexts, without an explicitly defined domain knowledge corpus. The absence of knowledge corpus prevents these benchmarks from conducting domain knowledge learning and modeling, confining their utility to performance evaluation and leaving the development of novel domain specialization approaches unsupported.

In this paper, we propose KoCo-Bench, a novel code benchmark designed to evaluate domain specialization methods for LLMs. Unlike existing benchmarks, KoCo-Bench innovatively provides knowledge corpora alongside corresponding test sets. The benchmark took 28.5 person-months to construct, covering 6 emerging domains: RL, Agent, RAG, Model Optimization, Embodied AI, and Ascend Ecosystem. The knowledge corpus is curated from multiple sources (*i.e.*, framework documentation, framework source code, and usage examples), simulating the introduction of new knowledge sources when LLMs are applied to develop projects based on unfamiliar frameworks. Based on knowledge corpus, we evaluate two tasks: domain code generation and domain knowledge understanding. For domain code generation, we provide multi-granularity requirement specifications (*i.e.*, project descriptions, module divisions, and core function descriptions) to support function-level to project-level generation, with rigorous verification through test suites (*i.e.*, unit tests and integration tests). For domain knowledge understanding, we use easily verifiable multiple-choice Q&A to evaluate whether LLMs can comprehend the knowledge within the corpus.

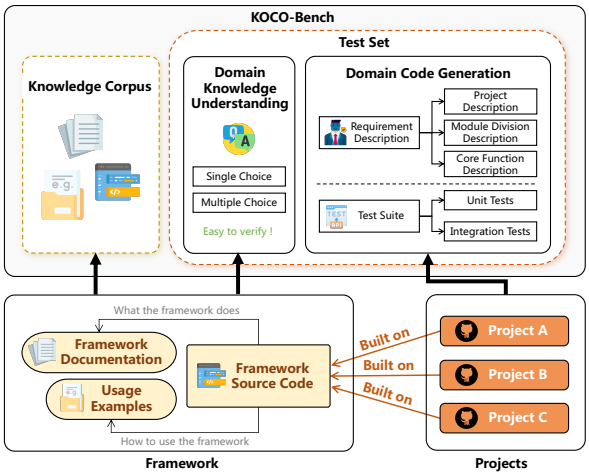

Figure 1: An overview of KoCo-Bench.

We conduct extensive experiments on KoCo-Bench, encompassing state-of-the-art LLMs (e.g., Claude Sonnet 4.5, Kimi-K2, Gemini 2.5 Pro), representative domain specialization methods (SFT, RAG, kNN-LM), and composite agent systems representing current best practices (Claude Code, SWE-Agent, OpenHands). Beyond standard benchmarking, we perform exploratory studies to investigate how knowledge corpus scale affects learning efficacy, whether continual learning across domains induces catastrophic forgetting, etc. Our experiments yield five key findings: ❶ Even SOTA closed-source LLMs struggle with domain code generation and achieve only moderate performance on domain knowledge understanding. ❷ Existing domain specialization methods provide only marginal improvements and exhibit inconsistent effectiveness across domains. ❸ Agentic approaches currently offer the most effective solution, yet substantial room for improvement remains. ❹

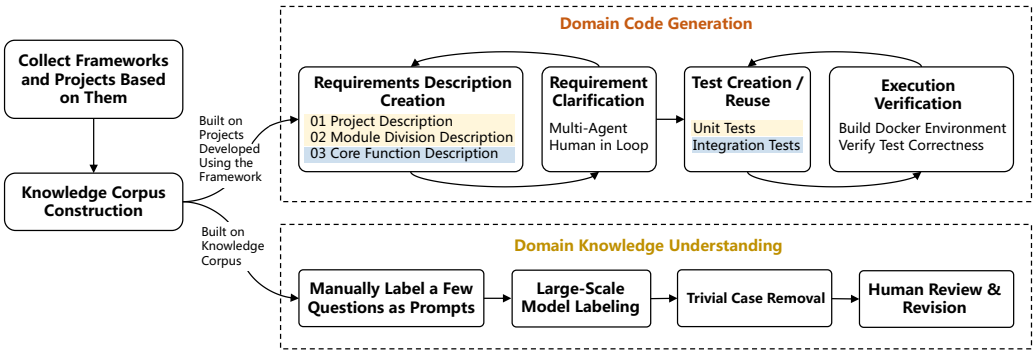

Figure 2: Construction pipeline of KoCo-Bench.

Learning-based domain specialization methods show diminishing effectiveness as the knowledge corpus grows larger, and continual learning across domains causes forgetting of previously acquired knowledge. ❺ The most prevalent errors in domain code generation are misusing domain-specific APIs and violating domain data constraints.

2 Related Work

2.1 Code Generation Benchmark

Existing code generation benchmarks fall into three categories. The first, general coding benchmarks, *e.g.*, HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), LiveCodeBench (Jain et al., 2024), primarily test syntax mastery and algorithmic reasoning, often sourced from competitive programming platforms. The second, library-oriented benchmarks, *e.g.*, DS-1000 (Lai et al., 2022), PandasEval and NumpyEval (Zan et al., 2022), evaluate the ability of LLMs to understand and apply specific library functions. The third, repository-level benchmarks, *e.g.*, RepoBench (Liu et al., 2023), CrossCodeEval (Ding et al., 2023), SWE-bench (Jimenez et al., 2024), focus on retrieval and context understanding for code generation and issue resolution. While some recent benchmarks, such as EvoCodeBench (Li et al., 2024), DomainCodeBench (Zheng et al., 2025), DomainEval (Zhu et al., 2024), and MultiCodeBench (Zheng et al., 2025), target domain-specific code generation, they only extract information from repository contexts, thus falling into this third category.

These benchmarks share a common limitation: they assume models either possess relevant domain knowledge or can extract it directly from given contexts. However, most domain knowledge is neither pre-existing in models nor available in contexts, but rather needs to be learned and understood from domain knowledge corpus. Without an associated knowledge corpus, existing benchmarks cannot support domain knowledge learning and modeling processes, limiting their value to performance evaluation rather than advancing domain specialization methods.

2.2 Domain Specialization Method For Software Development

To adapt LLMs for domain-specific software development, researchers have proposed various domain specialization methods. General approaches include fine-tuning (Dong et al., 2024a; Pareja et al., 2024), RAG (Gao et al., 2023; Lewis et al., 2020), and kNN-LM (Khandelwal et al., 2020). Fine-tuning updates model parameters through training on domain data, with parameter-efficient techniques like LoRA (Hu et al., 2021), QLoRA (Dettmers et al., 2023), and Adapter (Poth et al., 2023) developed to reduce computational costs. RAG retrieves relevant information from external knowledge bases during inference and injects it as context into prompts. kNN-LM retrieves k-nearest neighbors from (hidden state, token) datastores and fuses their output distributions with model predictions for knowledge injection. Researchers have adapted these methods for code domains. kNM-LM (Tang et al., 2023) improves kNN-LM by storing only mispredicted samples and using Bayesian inference for automatic weight calculation. DomCoder (Gu et al.) combines RAG with Chain-of-Thought reasoning to incorporate domain API knowledge. Other work proposes large-small model collaboration (Yu et al., 2025), where fine-tuned small models provide domain expertise and large models offer general capabilities, with a classifier selecting which generates each token.

However, the evaluation of these methods on

software development remains severely inadequate. Existing studies typically use self-constructed naive datasets, training on functions from the same-domain projects or within-project splits, then testing on the remaining functions for code completion. The incompleteness of this evaluation method is twofold. First, the knowledge overlap is weak among projects within the same domain, or even among functions within a single project, leaving knowledge reference seriously insufficient. Second, code completion tasks, which predict continuations from given contexts, have multiple valid answers, making evaluation inaccurate. Without proper consensus benchmarks, despite urgent demand for effective domain specialization methods, this research direction remains underdeveloped.

3 KoCo-Bench

KoCo-Bench benchmarks LLMs on acquiring and applying external knowledge to software development by simulating the scenario of developing new projects based on domain software frameworks. An overview of KoCo-Bench is presented in Figure 1.

3.1 Task Formulation

KoCo-Bench consists of two core components: Knowledge Corpus and Test Set. The Knowledge Corpus can be utilized through various approaches (e.g., fine-tuning, retrieval augmentation) to enhance the performance of LLMs on the test set. The Knowledge Corpus provides a domain knowledge foundation, including framework documentation, framework source code, and usage examples. The test Set covers the two most common tasks when developers use AI-assisted programming, *i.e.*, domain code generation and domain knowledge understanding: ❶ **The domain code generation task** takes the form of natural language requirement descriptions to code implementation, evaluating the correctness of generated code through test execution. We provide multi-level requirement information (including project descriptions, module division descriptions, and core function descriptions) and multi-level testing (including unit tests and integration tests), supporting code generation evaluation at different granularities from function-level to project-level, to accommodate future trends in code generation technology. ❷ **The domain knowledge understanding task (Q&A)** can specifically assess models' mastery of particular knowledge points. Existing code repository Q&A benchmarks (Hu et al., 2024; Liu and Wan, 2021) typically use GitHub Issues for construction and employ text similarity or LLM-as-judge for evaluation, suffering from the inability to precisely assess domain knowledge and inaccurate evaluation. Our Q&A task specifically targets domain knowledge assessment and adopts a multiple-choice format that can be precisely evaluated, ensuring the reliability of evaluation.

3.2 Benchmark Construction

This section details the construction process of KoCo-Bench, including the Knowledge Corpus, domain code generation tasks, and domain knowledge understanding tasks. To ensure quality while controlling costs, we follow previous work (Chi et al., 2025) and introduce an LLM agent (*i.e.*, Claude Code) to assist in certain annotation tasks, using it only within work scopes where its performance has been pre-evaluated and confirmed reliable by humans, with all agent-generated content requiring human review and correction. The construction pipeline of KoCo-Bench is illustrated in Figure 2.

3.2.1 Knowledge Corpus

Constructing a knowledge corpus that naturally aligns with evaluation tasks is typically challenging. We identified software frameworks and their implementations as an ideal scenario for this purpose. Software frameworks not only embody core abstractions, design patterns, and implementation logic but also codify domain knowledge and best practices at the code level. These frameworks usually come with comprehensive documentation and examples, forming essential knowledge components. Developers build various projects based on these frameworks, creating a natural ecosystem of knowledge reuse. Leveraging this natural knowledge organization, we construct our Knowledge Corpus from three sources: framework documentation, framework source code, and framework usage examples. This corpus then serves as the knowledge foundation for our evaluation tasks. We use projects built with these frameworks to create code generation tasks and design Q&A tasks to assess knowledge understanding and application.

For corpus construction, we first filter Python-based frameworks from GitHub, selecting those created since March 2024. We further require frameworks to have comprehensive documenta-

tion. We then select the top 10 frameworks by GitHub star count, spanning 5 different domains (*i.e.*, RL, Agent, RAG, Model Optimization, and Embodied AI). Additionally, we include two recently open-sourced underlying acceleration and support frameworks for Ascend AI processors (one Python framework and one C++ framework). The frameworks and projects involved are detailed in Table 7 (Appendix).

Table 1: Statistics of KoCo-Bench.

	Total Count	Granular Metrics	**Mean**	**Max**
Knowledge Corpus (Framework)	11	Count per Domain	1.8	4
		Length (Lines)	77.0k	400.5k
Projects	25	Count per Framework	2.3	7
		Length (Lines)	63.8k	674.0k
Core Functions	131	Count per Project	5.2	17
		Length (Lines)	52.1	346
Tests	978	Unit Tests per Core Function	8.6	24
		Integration Tests per Project	2.3	6
QA	107	Single-Choice Count per Frame	3.5	7
		Multi-Choice Count per Frame	14.3	15

3.2.2 Domain Code Generation

To ensure the completeness of the knowledge corpus, we select projects that are predominantly implemented based on their corresponding frameworks and do not include external dependencies created after the cutoff date. Based on these projects, we construct code generation tasks, involving the creation of requirement descriptions and test suites. This annotation work was completed by 7 annotators over a period of 2 months.

Requirements Description Creation. Requirements description is structured at three levels. Project Description provides an overview of project requirements, Module Division Description functions clarify the overall project structure, module responsibilities and interactions, while Core Function Description details the core functional methods of the project. When writing requirements, we ensure they focus on the problem domain rather than code implementation, select core functions related to domain knowledge while excluding utility classes and helper methods (such as config and utils), and avoid redundant or ambiguous non-functional requirements.

To ensure clarity and accuracy of requirements, we employ multi-agent assistance for requirement clarification. First, an agent attempts to write code based on the requirements to identify ambiguous descriptions. This step does not require agents to produce correct code, but rather to expose ambiguous requirements during the implementation process. The identified issues are then fed back to humans and another agent for review and requirement enhancement. This process iterates repeatedly until the requirements are confirmed as clear and complete through human verification.

Test Suite Creation. The test suite comprises unit tests, integration tests, and runtime virtual environments. Unit tests target core functions. If the project already contains unit tests for core functions, we reuse them directly. For functions without tests, we first write test inputs to cover all branches (validating the branch coverage using `coverage.py`), then run the ground-truth code to obtain expected outputs, and finally have an agent generate the test code. For functions without explicit inputs and outputs, we employ instrumentation to verify the correctness of function call sequences. Integration tests primarily reuse existing pipeline test scripts of the project or are manually written to verify the correctness of multi-module interactions. All tests must pass validation against the ground-truth code and are saved in the corresponding Docker environments.

3.2.3 Domain Knowledge Understanding

For frameworks where we could not identify appropriate projects for code generation benchmarks, such as Ascend-related frameworks, we construct domain knowledge understanding Q&A tasks. This work was completed by 3 annotators over 1.5 months.

The Q&A task design follows four principles. ❶ It only assesses domain-specific knowledge. ❷ Answers must be verifiable, so we adopt a multiple-choice format that allows skipping questions and supports multiple selections. ❸ Each question maintains atomicity by examining only one point while ensuring questions have unique solutions to avoid ambiguity. ❹ Question types are diversified, including complex questions that require synthesizing information from multiple files to answer.

The annotation process begins with humans writing desired, high-quality Q&A examples as prompts, then using an agent to generate at scale. Next, we filter trivial cases by testing with three LLMs [1], eliminating simple questions that all three models answer correctly, as we find these questions

[1]The three models we used are Qwen2.5-Coder-7B-Instruct, Llama-3.1-8B-Instruct, and Deepseek-Coder-7b, whose training data cutoff dates are all before KoCo-Bench's cutoff date to avoid data contamination.

Table 2: Comparison of Different Domain Code Evaluation Benchmarks.

Benchmark	Knowledge Corpus	Domain Code Generation					Domain Code Understanding
		Project Description	Module Division Description	Core Function Description	Unit Test	Integration Test	Q&A
DomainCodeBench	✗	✗	✗	✔	✗	✗	✗
DomainEval	✗	✗	✗	✔	✔	✗	✗
EvoCodeBench	✗	✗	✗	✔	✔	✗	✗
MultiCodeBench	✗	✗	✗	✔	✔	✗	✗
KoCo-bench	✔	✔	✔	✔	✔	✔	✔

typically lack distracting options and are easy to answer. Finally, we conduct a manual review and revision to ensure question quality. Through this rigorous quality control process, we ensure that the Q&A tasks can effectively evaluate models' understanding of domain knowledge.

3.3 Features of KoCo-Bench

Table 1 summarizes the statistics of KoCo-Bench, covering the scale and composition of the knowledge corpus, projects, tasks, and evaluation artifacts. Table 2 (Appendix) presents a comparison between KoCo-Bench and existing domain-specific code benchmarks. We highlight four key characteristics: ❶ **Realistic development scenarios.** KoCo-Bench is built from real-world software frameworks and projects. The natural alignment between knowledge corpora (frameworks) and evaluation tasks (projects built upon them) ensures that the benchmark reflects software engineering workflow. ❷ **Diverse domain knowledge.** KoCo-Bench covers 6 domains, including 11 knowledge corpora (frameworks) and 25 projects, showcasing substantial diversity. The knowledge corpora average over 76K lines per domain and reach up to 400K lines in the largest case, providing rich domain knowledge for evaluation. ❸ **Long-context and multi-granularity inputs.** Projects in test set average more than 63K lines of code, creating long-context scenarios. Together with the knowledge corpus and multi-level requirement specifications, KoCo-Bench provides diverse inputs, supporting a wide range of methods, including learning-based, retrieval-based, or hybrid methods. ❹ **Reliable evaluation.** Unlike benchmarks relying on textual similarity or LLM-based judges, KoCo-Bench adopts well-defined correctness criteria. Domain Code generation is evaluated via test suites (averaging 8.6 unit tests per function, 2.3 integration tests per project). Domain understanding tasks use single- and multiple-choice questions for unambiguous evaluation.

4 Experiments

In this section, we first assess the out-of-the-box performance of current LLMs on KoCo-Bench. We then examine whether existing domain specialization methods can effectively leverage the knowledge corpus to improve performance. Finally, we evaluate SOTA coding agents that integrate multiple techniques. Beyond these evaluations, we conduct in-depth analyses investigating knowledge forgetting during continual learning and the impact of corpus size on learning effectiveness. Additional experiments on common failure patterns (§A), project code generation (§B), data contamination detection (§H), and performance-cost tradeoffs (§I), along with case studies (§J), and detailed experiment setups (§K) are provided in Appendix.

4.1 Experiment Results

Performance of Various LLMs on KoCo-Bench. We evaluate 10 well-known LLMs on **KoCo-Bench**, including both closed-source and open-source models across different model families and parameter scales. We explicitly mark each model's training data cutoff date to ensure transparency for potential data contamination. The experimental results are presented in Table 3.

All LLMs exhibit notably low performance on KoCo-Bench. Even the best-performing models achieve only single-digit Pass@1 scores (Kimi-K2-Instruct: 8.9%, Gemini-2.5-pro: 8.5%), standing in stark contrast to their strong performance on general-purpose benchmarks such as HumanEval, where they exceed 90% Pass@1. For domain knowledge understanding, even the best LLMs achieve only 53.5% accuracy. This substantial performance gap underscores the inherent difficulty of domain-specific development and validates the necessity of KoCo-Bench as a challenging benchmark. Notably, all LLMs score zero on the RAG domain, as it requires domain-specific API calls that none of the models invoked correctly (manu-

Table 3: Performance of LLMs on KoCo-Bench across six emerging domains.

Cutoff Date	Models	Domain Code Generation										Domain Knowledge Understanding		
		RL Frameworks		Agent Frameworks		RAG Frameworks		MO Frameworks		Average		E-AI Frameworks	AE Frameworks	Average
		Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR	ACC	ACC	ACC
Jan-25	Gemini-2.5-pro	7.6	23.9	9.6	21.3	0.0	0.0	16.7	21.9	8.5	16.8	45.7	27.0	36.4
Jul-25	Claude-Sonnet-4-5	7.6	28.6	5.8	21.3	0.0	0.0	11.1	34.8	6.1	21.2	62.7	43.8	53.2
Sep-25*	Kimi-K2-Instruct	5.6	24.3	7.7	27.6	0.0	0.0	22.2	40.6	**8.9**	23.1	51.7	55.4	**53.5**
Aug-25*	DeepSeek-V3.1	5.6	20.8	7.7	27.4	0.0	0.0	11.1	34.8	6.1	20.7	49.5	34.7	42.1
May-24	GPT-5-mini	7.8	37.4	7.7	26.3	0.0	0.0	11.1	37.6	6.6	**25.3**	55.8	48.6	52.2
Jun-24	o4-mini	5.8	22.9	5.8	18.7	0.0	0.0	16.7	29.4	7.1	17.8	48.5	47.7	48.1
LLMs trained before KoCoBench ↓														
Feb-24	Qwen2.5-Coder-32B-Instruct	5.8	11.4	3.9	25.0	0.0	0.0	11.1	22.7	5.2	14.8	38.2	36.9	37.5
Feb-24	Qwen2.5-Coder-7B-Instruct	3.8	11.4	3.9	17.1	0.0	0.0	22.2	33.4	7.5	15.5	17.6	21.6	19.6
Dec-23	Llama-3.1-8B-Instruct	1.8	9.4	1.9	16.6	0.0	0.0	16.7	29.4	5.1	13.8	17.3	29.3	23.3
Feb-23	Deepseek-Coder-7B	1.8	3.2	0.0	0.6	0.0	0.0	11.1	13.0	3.2	4.2	5.0	2.3	3.6

1. The six emerging domains include Reinforcement Learning (RL), Agent, Retrieval-Augmented Generation (RAG), Model Optimization (MO), Embodied AI (E-AI), and Ascend Ecosystem (AE).
2. The Cutoff Date refers to the training data cutoff of each LLM by default. A Cutoff Date marked with '*' indicates the release date of LLM, as the training data cutoff is not publicly available.
3. While baselines score zero in RAG domain, Claude Code's 62.5% Pass@1 (Table 5) confirms these tasks are solvable via proper domain knowledge access.

Table 4: Performance of domain specialization methods on KoCo-Bench across six emerging domains.

Method	**Domain Code Generation**										**Domain Knowledge Understanding**		
	RL Frameworks		Agent Frameworks		RAG Frameworks		MO Frameworks		Average		E-AI Frameworks	AE Frameworks	Average
	Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR	ACC	ACC	ACC
Base Model	3.8	11.4	3.9	17.1	0.0	0.0	22.2	33.4	7.5	15.5	17.6	21.6	19.6
SFT	7.3	12.9	1.9	16.1	0.0	0.0	16.7	24.4	6.5	13.4	25.6	33.8	29.7
LoRA	5.5	16.1	3.9	14.0	0.0	0.0	16.7	21.9	6.5	13.0	17.9	35.5	26.7
RAG	7.2	23.7	5.8	12.8	0.0	0.0	16.7	38.9	7.4	18.9	25.1	35.5	30.3
kNN-LM	5.4	12.3	1.9	16.1	0.0	0.0	16.7	25.0	6.0	13.3	21.5	29.3	25.4

1. The six domains include Reinforcement Learning (RL), Agent, Retrieval-Augmented Generation (RAG), Model Optimization (MO), Embodied AI (E-AI), and Ascend Ecosystem (AE).
2. While baselines score zero in RAG domain, Claude Code's 62.5% Pass@1 (Table 5) confirms these tasks are solvable via proper domain knowledge access.

ally verified). Experiments in Table 5 show that Claude Code achieves 62.5% Pass@1, confirming the task is solvable with proper domain knowledge access. A counterintuitive result emerges in the Model Optimization domain, where larger LLMs sometimes underperform smaller ones. Our analysis reveals that larger LLMs are more prone to invoke external APIs, many of which do not exist or are incompatible with the specified runtime. These results highlight the urgent need for domain specialization methods.

Evaluation of Domain Specialization Methods. We evaluate various domain specialization methods on KoCo-Bench. We adopt Qwen2.5-Coder-7B-Instruct as the base model, whose training data cutoff predates our dataset's cutoff, thereby avoiding potential data leakage. Specifically, we assess four representative general methods: SFT, LoRA, RAG, and kNN-LM. These methods differ in how they incorporate domain knowledge: SFT and LoRA are training-based approaches that update model parameters, RAG is an inference-time retrieval method, and kNN-LM combines both training (datastore construction) and inference (retrieval). The results in Table 4 reveal that existing domain specialization methods struggle with domain code generation. Each method exhibits varying effectiveness across domains, achieving gains in some while incurring losses in others. For domain knowledge understanding, all methods show improvements, yet the best accuracy reaches only 30.3%. Among all methods, RAG performs best, followed by SFT. We further evaluate two code-specific methods (KNM-LM and DSCC), originally designed for code completion tasks. We carefully adapt them to code generation, with results in Appendix Table 8. Both methods underperform the base model: Pass@1 shows nearly no improvement, and APR gains are marginal, with performance drops in certain domains. This may be attributed to the difficulty of adapting completion-oriented methods to generation tasks with rigorous test validation. These findings underscore the urgent need for effective, code-oriented domain specialization methods.

Performance of LLM Agents on KoCo-Bench. We also evaluate agent methods that combine multiple techniques (*e.g.*, search, tool use, and mem-

Table 5: Performance of open-source and closed-source agents on KOCO-BENCH across six emerging domains.

Method	RL Frameworks		Agent Frameworks		RAG Frameworks		MO Frameworks		Average		Token Cost
	Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR	
Base Model											
Qwen2.5-Coder-32B-Instruct	5.8	11.4	3.9	25.0	0.0	0.0	11.1	22.7	5.2	14.8	1,247
Open-source Agent											
SWE-agent	3.6	6.6	1.9	12.6	0.0	0.0	12.5	22.5	4.5	10.4	26,583
OpenHands	1.8	4.8	0.0	0.0	0.0	2.5	12.5	14.4	3.6	5.4	26,005
Base Model											
Claude-Sonnet-4-5	7.6	28.6	5.8	21.3	0.0	0.0	11.1	34.8	6.1	21.2	1,760
Closed-source Agent											
Claude Code	13.6	26.4	38.5	56.3	62.5	68.8	22.2	45.6	34.2	49.3	619,923

1. The six domains include Reinforcement Learning (RL), Agent, Retrieval-Augmented Generation (RAG), Model Optimization (MO), Embodied AI (E-AI), and Ascend Ecosystem (AE).
2. Token Cost denotes the average number of tokens generated per sample, including both input and output tokens, for each baseline.

ory) to further advance the effectiveness of domain knowledge utilization. We assess two representative open-source coding agents (SWE-Agent (Yang et al., 2024) and OpenHands (Wang et al., 2025)) and one popular closed-source coding agent tool (Claude Code (ClaudeCode)) on the domain code generation task to investigate the performance ceiling achievable by SOTA techniques on KOCO-BENCH. The open-source agents are based on Qwen2.5-Coder-32B-Instruct, while Claude Code is powered by Claude-Sonnet-4-5. All agent methods operate with full access to the project folder and knowledge corpus, performing iterative exploration, code generation, and execution-validation cycles. The results are presented in Table 5.

Claude Code significantly outperforms SWE-Agent and OpenHands, achieving a Pass@1 of 34.2%. Notably, Claude Code attains a Pass@1 of 62.5% in the RAG domain, effectively solving code generation tasks that other domain specialization methods fail to address. We analyze three primary factors contributing to the limited effectiveness of SWE-Agent and OpenHands: 1) the base model (Qwen2.5-Coder-32B-Instruct) has not been specifically trained for agentic coding scenarios, resulting in weaker agentic capabilities; 2) both agents are prone to instruction drift, gradually losing track of the original repo-level task requirements and function signatures during extended interactions; and 3) these agents frequently fall into repetitive loops of tool invocations when attempting to gather contextual information, wasting computational resources without making progress. Moreover, agentic methods suffer from high inference costs, consuming more tokens than other approaches. To address this confound, we conduct control experiments with Claude-Sonnet-4-5 as the base model for all agents (§G). The results show that switching to the stronger base model yields notable improvements with both SWE-Agent and OpenHands, yet their performance still falls substantially short of Claude Code, revealing a gap between current open-source and closed-source agents.

Table 6: Impact of continual learning on knowledge.

	Single Framework	Multi-Framework Same Domain	Multi-Framework Cross-Domain
Pass@1	7.1	7.1	3.6
AvgPassRate	12.8	13.5	11.1

4.2 Further Analysis

Investigating Knowledge Forgetting and Conflict. In real-world deployment, models often continue learning across multiple frameworks or domains. A critical question is whether such sequential knowledge acquisition leads to forgetting or knowledge conflicts that degrade performance on previously learned content. We investigate this problem on KOCO-BENCH using SFT, and the results are presented in Table 6. When sequentially training on two frameworks within the same domain, the model maintains comparable performance to the single-framework baseline, with AvgPassRatio even showing a slight improvement. In contrast, cross-domain sequential training leads to notable performance degradation, with Pass@1 dropping from 7.1% to 3.6%. These results indicate that knowledge forgetting or conflict is largely determined by the semantic relationship between training corpora. When frameworks are closely related, joint training can yield synergistic benefits through shared representations. However, when frameworks originate from distant domains, sequential learning is more likely to cause knowledge overwriting, resulting in degraded performance on

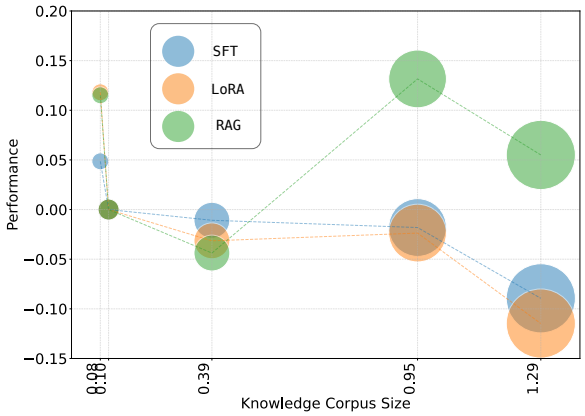

Figure 3: The impact of knowledge corpus size on domain specialization performance.

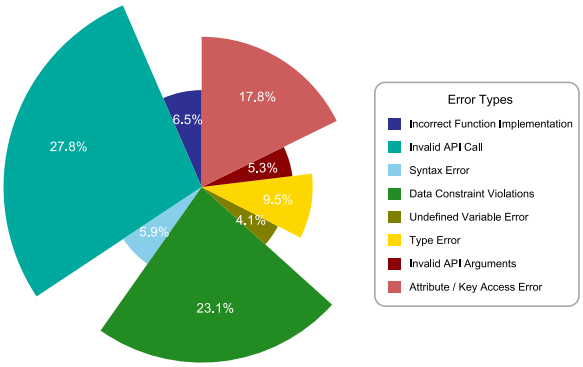

Figure 4: Error type distribution of the base model on KoCo-Bench.

previously learned content. This finding suggests that practitioners should exercise caution when extending models across domains, and that incorporating data replay strategies during training may be necessary to preserve prior knowledge.

Impact of Knowledge Corpus Size. Understanding the relationship between knowledge corpus size and learning effectiveness can provide insights into data requirements for domain specialization. We analyze this relationship on KoCo-Bench using SFT, LoRA, and RAG. Figure 3 presents this analysis, where the x-axis represents knowledge corpus size measured in millions of tokens (the standard metric for LLM input), the y-axis shows AvgPassRate on the corresponding domain code generation tasks, and the point size indicates Lines of Code (LOC), a traditional software engineering metric for codebase scale. The results reveal distinct scaling behaviors across methods. Both SFT and LoRA exhibit a negative correlation between corpus size and performance, indicating that incorporating more domain-specific data paradoxically hinders learning effectiveness. In contrast, RAG demonstrates no such correlation, with its performance remaining largely unaffected by corpus size.

Error Analysis. To identify the key challenges posed by KoCo-Bench, we categorize and analyze the errors produced by base model. The distribution is presented in Figure 4 (Appendix). The largest error category is Invalid API Call, where LLMs hallucinate non-existent APIs. Combined with Invalid API Arguments, API-related errors account for approximately one-third of all failures. This indicates that current LLMs lack sufficient knowledge of framework-specific APIs, often falling back on generic patterns that do not align with the design of the framework. Data Constraint Violations represent the second largest category, where generated code fails to adhere to expected data formats, value ranges, or structural requirements. Together with Attribute/Key Access Error, Type Error, and Undefined Variable Error, these errors reveal that LLMs struggle to reason about the constraints governing data flow in domain-specific code. Other error types, such as Syntax Error and Incorrect Function Implementation represent a smaller share of the overall distribution.

5 Conclusion

We present KoCo-Bench, a benchmark that innovatively pairs domain knowledge corpora with evaluation tasks to assess how effectively LLMs can acquire and apply new domain knowledge, which enables systematic evaluation across domain code generation and knowledge understanding tasks, spanning 6 emerging domains with 11 frameworks. Our experiments reveal that even SOTA LLMs struggle with domain code generation, and existing domain specialization methods offer only marginal improvements. Learning-based approaches suffer from diminishing returns as the knowledge corpus scales and exhibit catastrophic forgetting during continual learning, while agentic methods still remain far from meeting practical domain requirements. These findings highlight domain-specific software development as a significant challenge for LLMs, and we hope KoCo-Bench will catalyze future research toward effective solutions.

Limitations

Our work has the following two limitations: First, since LLMs continue to evolve, we take proactive

measures to mitigate data contamination risks and ensure reliable evaluation. We explicitly establish a dataset cutoff date and select frameworks created after this date. When evaluating domain specialization methods, we deliberately use base models whose training cutoff precedes the dataset's cutoff. Additionally, we conduct contamination detection (detailed in Appendix H), which demonstrates negligible contamination probability. The challenging nature of KoCo-Bench, where even SOTA models achieve limited performance, further validates the robustness of our benchmark against potential data contamination concerns for domain specialization methods. Second, constrained by API availability and access limitations at the time of our experiments, we do not evaluate several recently released LLMs, such as OpenAI GPT-5 and Gemini 3 Pro. We will extend our evaluation to cover the emerging models in future updates of KoCo-Bench.

Acknowledgments

We thank Lecheng Wang and Liye Zhu for their assistance with the preliminary domain investigation. We are grateful to Zejun Wang for participation in discussions regarding test suite creation. We also acknowledge every member of the SEKE Team for their valuable suggestions provided during team discussions. This research is supported by the National Natural Science Foundation of China under Grant No. 62192733, 62192730, 62192731, the National Key R&D Program under Grant No. 2023YFB4503801, and the Beijing Major Science and Technology Project under Contract No. Z251100008425005.

References

Garima Agrawal, Tharindu Kumarage, Zeyad Alghamdi, and Huan Liu. 2024. Mindful-rag: A study of points of failure in retrieval augmented generation. *Preprint*, arXiv:2407.12216.

Anthropic. 2025. Claude sonnet. https://www.anthropic.com/claude/sonnet. Official Anthropic documentation.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program synthesis with large language models. *Preprint*, arXiv:2108.07732.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Wayne Chi, Valerie Chen, Ryan Shar, Aditya Mittal, Jenny Liang, Wei-Lin Chiang, Anastasios Nikolas Angelopoulos, Ion Stoica, Graham Neubig, Ameet Talwalkar, and Chris Donahue. 2025. Edit-bench: Evaluating LLM abilities to perform real-world instructed code edits. *CoRR*, abs/2511.04486.

ClaudeCode. 2025. ClaudeCode. https://www.anthropic.com/claude-code.

deepseek ai. 2025. deepseek-coder-7b-base-v1.5. https://huggingface.co/deepseek-ai/deepseek-coder-7b-base-v1.5. Hugging Face model card.

deepseek-ai. 2025. Deepseek-v3.1. https://huggingface.co/deepseek-ai/DeepSeek-V3.1. Hugging Face model card.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. *Preprint*, arXiv:2305.14314.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. 2023. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. *Preprint*, arXiv:2310.11248.

Guanting Dong, Hongyi Yuan, Keming Lu, Chengpeng Li, Mingfeng Xue, Dayiheng Liu, Wei Wang, Zheng Yuan, Chang Zhou, and Jingren Zhou. 2024a. How abilities in large language models are affected by supervised fine-tuning data composition. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 177–198.

Yihong Dong, Xue Jiang, Huanyu Liu, Zhi Jin, Bin Gu, Mengfei Yang, and Ge Li. 2024b. Generalization or memorization: Data contamination and trustworthy evaluation for large language models. *Preprint*, arXiv:2402.15938.

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, Haofen Wang, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2(1).

Gemini3. 2025. Gemini3. https://deepmind.google/models/gemini/.

Sreyan Ghosh, Chandra Kiran Reddy Evuru, Sonal Kumar, Ramaneswaran S, Deepali Aneja, Zeyu Jin, Ramani Duraiswami, and Dinesh Manocha. 2024. A closer look at the limitations of instruction tuning. *Preprint*, arXiv:2402.05119.

GPT-5. 2025. GPT-5. https://openai.com/zh-Hans-CN/gpt-5/.

Xiaodong Gu, Meng Chen, Yalan Lin, Yuhan Hu, Hongyu Zhang, Chengcheng Wan, Zhao Wei, Yong Xu, and Juhong Wang. On the effectiveness of large language models in domain-specific code generation. *ACM Transactions on Software Engineering and Methodology*, 34(3).

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. Lora: Low-rank adaptation of large language models. *Preprint*, arXiv:2106.09685.

Ruida Hu, Chao Peng, Jingyi Ren, Bo Jiang, Xiangxin Meng, Qinyun Wu, Pengfei Gao, Xinchen Wang, and Cuiyun Gao. 2024. Coderepoqa: A large-scale benchmark for software engineering question answering. *Preprint*, arXiv:2412.14764.

Chia-Chien Hung, Wiem Ben Rim, Lindsay Frost, Lars Bruckner, and Carolin Lawrence. 2023. Walking a tightrope – evaluating large language models in high-risk domains. *Preprint*, arXiv:2311.14966.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2024. Livecodebench: Holistic and contamination free evaluation of large language models for code. *Preprint*, arXiv:2403.07974.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? *Preprint*, arXiv:2310.06770.

Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. 2020. Generalization through memorization: Nearest neighbor language models. *Preprint*, arXiv:1911.00172.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. 2022. Ds-1000: A natural and reliable benchmark for data science code generation. *Preprint*, arXiv:2211.11501.

Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, and 1 others. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems*, 33:9459–9474.

Jia Li, Ge Li, Xuanming Zhang, Yihong Dong, and Zhi Jin. 2024. Evocodebench: An evolving code generation benchmark aligned with real-world code repositories. *Preprint*, arXiv:2404.00599.

Xiaofeng Lin, Hejian Sang, Zhipeng Wang, and Xuezhou Zhang. 2025. Debunk the myth of sft generalization. *Preprint*, arXiv:2510.00237.

Chenxiao Liu and Xiaojun Wan. 2021. Codeqa: A question answering dataset for source code comprehension. *Preprint*, arXiv:2109.08365.

Tianyang Liu, Canwen Xu, and Julian McAuley. 2023. Repobench: Benchmarking repository-level code auto-completion systems. *Preprint*, arXiv:2306.03091.

meta-llama. 2024. Llama 3.1 8b. https://huggingface.co/meta-llama/Llama-3.1-8B. Hugging Face model card.

moonshotai. 2025. Kimi-k2-instruct-0905. https://huggingface.co/moonshotai/Kimi-K2-Instruct-0905. Hugging Face model card.

OpenAI. 2025a. Gpt-5 mini. https://platform.openai.com/docs/models/gpt-5-mini. Official OpenAI documentation.

OpenAI. 2025b. o4-mini model. https://platform.openai.com/docs/models/o4-mini. Official OpenAI documentation.

Aldo Pareja, Nikhil Shivakumar Nayak, Hao Wang, Krishnateja Killamsetty, Shivchander Sudalairaj, Wenlong Zhao, Seungwook Han, Abhishek Bhandwaldar, Guangxuan Xu, Kai Xu, Ligong Han, Luke Inglis, and Akash Srivastava. 2024. Unveiling the secret recipe: A guide for supervised fine-tuning small llms. *Preprint*, arXiv:2412.13337.

Clifton Poth, Hannah Sterz, Indraneil Paul, Sukannya Purkayastha, Leon Engländer, Timo Imhof, Ivan Vulić, Sebastian Ruder, Iryna Gurevych, and Jonas Pfeiffer. 2023. Adapters: A unified library for parameter-efficient and modular transfer learning. *Preprint*, arXiv:2311.11077.

Qwen. 2024a. Qwen2.5-coder-32b-instruct. https://huggingface.co/Qwen/Qwen2.5-Coder-32B-Instruct. Hugging Face model card.

Qwen. 2024b. Qwen2.5-coder-7b-instruct. https://huggingface.co/Qwen/Qwen2.5-Coder-7B-Instruct. Hugging Face model card.

Ze Tang, Jidong Ge, Shangqing Liu, Tingwei Zhu, Tongtong Xu, Liguo Huang, and Bin Luo. 2023. Domain adaptive code completion via language models and decoupled domain databases. *Preprint*, arXiv:2308.09313.

Xingyao Wang, Simon Rosenberg, Juan Michelini, Calvin Smith, Hoang Tran, Engel Nyst, Rohit Malhotra, Xuhui Zhou, Valerie Chen, Robert Brennan, and Graham Neubig. 2025. The openhands software agent sdk: A composable and extensible foundation for production agents. *Preprint*, arXiv:2511.03690.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Preprint*, arXiv:2405.15793.

Jingrong Yu, Zhipeng Gao, Lingfeng Bao, and Zhongxin Liu. 2025. Enhancing domain-specific code completion via collaborative inference with large and small language models. *ACM Transactions on Software Engineering and Methodology*.

Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. 2022. Cert: Continual pre-training on sketches for library-oriented code generation. *Preprint*, arXiv:2206.06888.

Qinggang Zhang, Shengyuan Chen, Yuanchen Bei, Zheng Yuan, Huachi Zhou, Zijin Hong, Hao Chen, Yilin Xiao, Chuang Zhou, Junnan Dong, Yi Chang, and Xiao Huang. 2025. A survey of graph retrieval-augmented generation for customized large language models. *Preprint*, arXiv:2501.13958.

Dewu Zheng, Yanlin Wang, Ensheng Shi, Xilin Liu, Yuchi Ma, Hongyu Zhang, and Zibin Zheng. 2025. Top general performance = top domain performance? domaincodebench: A multi-domain code generation benchmark. *Preprint*, arXiv:2412.18573.

Qiming Zhu, Jialun Cao, Yaojie Lu, Hongyu Lin, Xianpei Han, Le Sun, and Shing-Chi Cheung. 2024. Domaineval: An auto-constructed benchmark for multi-domain code generation. *Preprint*, arXiv:2408.13204.

A Structure and Statistics of KoCo-Bench

Table 7 summarizes the hierarchical structure of KoCo-Bench. The benchmark is organized into two tasks: Domain Code Generation and Domain Knowledge Understanding. These tasks cover diverse domains including Reinforcement Learning, Agent, RAG, Model Optimization, Embodied AI, and Ascend Ecosystem. Each domain is grounded in one or more representative frameworks, and for the code generation task, we curate real-world projects built on these frameworks as evaluation targets.

Table 7: Structure and statistics of KoCo-Bench across tasks, domains, and frameworks.

Tasks	**Domain Code Generation**					**Domain Knowledge Understanding**	
Domain	Reinforcement Learning		Agent	RAG	Model Optimization	Embodied AI	Ascend Ecosystem
Framework	VeRL	Open-R1	Smolagents	RAG-anything	TensorRT	VSLAM-LAB, cosmos-rl, robocasa, trackerLab	ascend-transformer-boost, triton-ascend
Project	prime, PURE, ARES, LUFFY, DAPO, PACS, critic-rl	AlphaDrive, CANOE, VLM-R1, VisualQuality-R1, mlldiffucoder	DeepSearchAgents, ToolBrain, examples, smolcc	BookWorm, CHAT-ANYTHING, law-unleashed-rag, obsidian-GraphRAG, rag4chat	FlagScale, Nemo, TensorRT-Incubator, byteperf	–	–

B Performance Analysis of Project Code Generation

Given the high task complexity and prohibitive computational cost, we perform a qualitative analysis on the project-level code generation task of KoCo-Bench. Given the absence of established methods for truly project code generation, we adopt Claude Code (based on Claude Sonnet 4.5), a state-of-the-art coding agent, augmented with the VeRL framework as an explicit domain knowledge base, to tackle this task. Given the project description and module division specification, the agent is tasked with generating a complete project that passes integration tests.

After generation, we execute the produced project end-to-end using a faithful debugging protocol: runtime errors are logged, minimally fixed, and the project is re-executed until the training pipeline runs successfully. This procedure allows us to analyze the types of errors that arise during execution.

We observe that most failures stem from a misalignment between the generated code and implicit domain knowledge. While the agent successfully reproduces the overall project structure and reuses framework-style components, it frequently violates unstated assumptions that are only enforced at runtime. These failures can be categorized into two main types:

- **Interface Misalignment.** The agent generates components that are individually correct but incompatible when composed together. This includes mismatches between data formats, configuration semantics, and module interfaces, suggesting that the agent lacks understanding of cross-module contracts within the framework.

- **Execution Semantics Misunderstanding.** The agent produces code that is syntactically valid but violates the runtime execution logic of the framework, particularly in training dynamics. This indicates that the agent fails to capture how different components interact during actual execution.

These findings suggest that the agent relies primarily on surface-level pattern imitation rather than a deep understanding of the execution logic of the framework. Effective project-level code generation requires not only structural knowledge of the framework, but also implicit knowledge about runtime behaviors and cross-module dependencies. Advancing project-level code generation remains an open challenge, and we hope KoCo-Bench provides a meaningful benchmark for future research in this direction.

C Performance of Code-specific Domain Specialization Methods

Since KNM-LM (Tang et al., 2023) and DSCC (Yu et al., 2025) are originally designed for code completion, we carefully adapt them to code generation for evaluation. Results are shown in Appendix Table 8.

Table 8: Performance of domain specialization methods for code on KOCO-BENCH.

Method	RL		Agent		RAG		Model Optimization		Average	
	Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR
Base Model	3.8	11.4	3.9	17.1	0.0	0.0	22.2	33.4	7.5	15.5
kNM-LM	1.8	3.3	0.0	0.0	0.0	0.0	5.6	10.5	1.8	3.4
DSCC	3.6	11.1	1.9	17.5	0.0	0.0	22.2	37.1	6.9	13.1

D Additional Domain Specialization Baselines

To further investigate the effectiveness of domain specialization methods beyond those reported in the main paper, we evaluate additional baselines including dense retrieval-based RAG and instruction-tuned fine-tuning with synthesized training pairs.

Dense RAG. We replace the BM25 lexical retriever with an embedding-based dense retriever (EmbeddingGemma), keeping all other RAG configurations identical to the original RAG baseline.

Instruction-tuned Fine-tuning. Instead of training on raw corpus text via next-token prediction, we use Gemini-3-Flash to generate code generation requirements for functions from our knowledge corpus, creating training pairs of (requirement, code) for instruction-tuned fine-tuning.

The results are presented in Table 9. For reference, we also include the original SFT and RAG baselines from the main paper. These additional approaches yield marginal improvements, with average performance across domains remaining below 10% Pass@1, which does not affect our original findings that existing domain specialization methods provide only limited gains.

Table 9: Performance of additional domain specialization baselines on KOCO-BENCH.

Method	RL		Agent		RAG		Model Optimization		Average	
	Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR
SFT	7.3	12.9	1.9	16.1	0.0	0.0	16.7	24.4	6.5	13.4
RAG (BM25)	7.2	23.7	5.8	12.8	0.0	0.0	16.7	38.9	7.4	18.9
Dense RAG	7.3	18.4	5.8	12.4	0.0	0.0	22.2	45.0	8.8	18.9
Instruction-tuned	5.4	15.4	13.5	35.2	0.0	0.0	16.7	27.0	8.9	19.4

E Ablation on Maximum Sequence Length

The default SFT configuration uses a maximum sequence length of 2048 tokens, while the knowledge corpora average approximately 77K lines. To verify whether the maximum sequence length is a determining factor, we conduct an ablation study on the SFT baseline with three different maximum sequence lengths: 2048, 4096, and 8192. As shown in Table 10, the results demonstrate that maximum sequence length has minimal impact on SFT performance and is not a determining factor for knowledge acquisition.

F Pass@any Results

To explore the performance upper bound, we report Pass@any results, which considers a sample correct if at least one of 10 attempts (with temperature=0.8) produces a correct solution. Table 11 presents Pass@any results for various LLMs, and Table 12 presents results for domain specialization methods. Pass@any shows moderate improvements over Pass@1, providing insights into the performance ceiling.

Table 10: Ablation study on maximum sequence length for SFT on KoCo-Bench.

Method	RL		Agent		RAG		Model Optimization		Average	
	Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR
SFT-2048	7.3	12.9	1.9	16.1	0.0	0.0	16.7	24.4	6.5	13.4
SFT-4096	5.6	15.7	1.9	19.9	0.0	0.0	16.7	29.8	6.0	16.3
SFT-8192	3.8	15.4	1.9	19.9	0.0	0.0	22.2	30.6	7.0	16.5

Table 11: Pass@any results of LLMs on KoCo-Bench (domain code generation).

Models	RL		Agent		RAG		Model Optimization		Average	
	Pass@1	Pass@any	Pass@1	Pass@any	Pass@1	Pass@any	Pass@1	Pass@any	Pass@1	Pass@any
Claude-Sonnet-4-5	8.2	9.8	12.7	17.3	0.0	0.0	16.7	16.7	9.4	10.9
Kimi-K2-Instruct	8.4	14.9	14.8	19.2	0.0	0.0	19.4	22.2	10.7	14.1
DeepSeek-V3.1	5.8	11.4	14.2	19.2	0.0	0.0	16.7	16.7	9.2	11.8
GPT-5-mini	10.2	13.8	13.1	19.2	0.0	0.0	16.7	16.7	10.0	12.4
Qwen2.5-Coder-32B-Instruct	1.8	1.8	3.7	9.6	0.0	0.0	5.6	5.6	2.7	4.2
Qwen2.5-Coder-7B-Instruct	2.9	7.6	3.7	3.9	0.0	0.0	14.4	22.2	5.3	8.4
Llama-3.1-8B-Instruct	3.6	11.6	5.4	15.4	0.0	0.0	16.7	16.7	6.4	10.9

Pass@1 values here are computed from temperature=0.8 sampling (10 samples), which may differ slightly from the greedy-decoded Pass@1 in Table 3.

G Control Experiments for Agent Evaluation

In Table 5, we controlled for the base model across open-source agents (using Qwen2.5-Coder-32B-Instruct for both SWE-Agent and OpenHands, whose training data cutoff predates our dataset's cutoff date to ensure contamination-free evaluation), while Claude Code is powered by Claude-Sonnet-4-5. Since Claude Code's terms of service restrict its use to Claude models only, this introduces a confounding factor. To address this confound, we conduct control experiments: Claude-Sonnet-4-5 + SWE-Agent and Claude-Sonnet-4-5 + OpenHands. The results are presented in Table 13.

The results show that switching to the stronger Claude-Sonnet-4-5 base model yields notable improvements with both SWE-Agent and OpenHands, yet their performance still falls substantially short of Claude Code. These results reveal a gap between current open-source agents and closed-source agents, indicating that further optimization is needed for open-source agents, which does not affect our original conclusion.

H Data Contamination Detection in KoCo-Bench

To verify whether our dataset suffers from data contamination, we apply CDD (Dong et al., 2024b), a widely adopted contamination detection method, to the base models evaluated in our experiments. The contamination index quantifies the likelihood that an LLM has been exposed to benchmark data during training, ranging from 0 to 1, where lower values indicate less contamination. We conduct detection on both Knowledge Corpus and Test Set across all domains, with results shown in Table 14. The average contamination index for Knowledge Corpus is 0.08, while that for Test Set is as low as 0.005, providing strong evidence that our benchmark is free from data contamination.

I Performance-Cost Trade-off Analysis

To provide a comprehensive understanding of the trade-offs between effectiveness and efficiency, we analyze both the training and inference costs of knowledge learning and utilization methods, as well as coding agents. Figure 5 illustrates the performance-cost landscape of different approaches. The y-axis represents performance (measured by Pass@1), while the x-axis shows inference token consumption. The size of each scatter point indicates the training time required by the method. This analysis provides actionable guidance for practitioners to identify suitable methods given their specific resource constraints.

Table 12: Pass@any results of domain specialization methods on KOCO-BENCH (domain code generation).

Method	RL		Agent		RAG		Model Optimization		Average	
	Pass@1	Pass@any	Pass@1	Pass@any	Pass@1	Pass@any	Pass@1	Pass@any	Pass@1	Pass@any
Base Model	2.9	7.6	3.7	3.9	0.0	0.0	14.4	22.2	5.3	8.4
SFT	3.4	7.3	7.3	17.3	0.0	0.0	15.6	22.2	6.6	11.7
LoRA	1.8	7.2	5.8	11.5	0.0	0.0	14.4	22.2	5.5	10.2
RAG	4.0	9.1	6.5	21.2	0.0	0.0	17.2	27.8	6.9	14.5
kNN-LM	5.2	5.4	6.5	9.6	0.0	0.0	16.7	16.7	7.1	7.9

Pass@1 values here are computed from temperature=0.8 sampling (10 samples), which may differ slightly from the greedy-decoded Pass@1 in Table 4.

Table 13: Control experiments: performance of agents with Claude-Sonnet-4-5 as the base model on KOCO-BENCH.

Method	RL		Agent		RAG		Model Optimization		Average	
	Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR	Pass@1	APR
Claude-Sonnet-4-5 (Base)	7.6	28.6	5.8	21.3	0.0	0.0	11.1	34.8	6.1	21.2
SWE-Agent	9.1	26.2	28.9	46.2	25.0	25.0	11.1	25.1	18.5	30.6
OpenHands	10.9	28.3	17.3	30.5	25.0	25.0	16.7	34.2	17.5	29.5
Claude Code	13.6	26.4	38.5	56.3	62.5	68.8	22.2	45.6	34.2	49.3

J Case Study

We provide two cases to illustrate the most frequent error types identified in our error analysis: Invalid API Call and Data Constraint Violations.

Case 1: Invalid API Call. Figure 6 presents an example of an Invalid API Call error. In this case, the model hallucinates a series of non-existent APIs, including `compute_kl_divergence`, `compute_abs_kl_divergence`, `compute_mse_kl_divergence`, `compute_low_var_kl_divergence`, and `compute_full_kl_divergence`. This error arises because the model is unfamiliar with the `FixedKLController` class defined in the knowledge corpus, and instead fabricates plausible-looking but invalid method names based on surface-level patterns.

Case 2: Data Constraint Violations. Figure 7 illustrates an example of a Data Constraint Violations error. In this case, the model is unaware of the implicit convention of the framework that `rm_scores` operates at token-level granularity while `acc` is defined at sequence-level. Due to this lack of understanding, the model incorrectly fuses rewards of different granularities as homogeneous objects and projects them onto an inconsistent temporal scale for RLOO computation, resulting in semantic errors during execution.

K Additional Implementation Details

K.1 Experiment Setup

Domain Code Generation. We primarily evaluate the generation of core functions. Since generating entire projects proves extremely challenging (which we present in Appendix B), we focus on core function generation as the main evaluation target. For direct LLM generation, we provide the core function description and function signature as input, prompting models to generate the function body. For knowledge learning and utilization methods, we distinguish between two scenarios: 1) Inference-based methods can access both the knowledge corpus and the instance project context, but we carefully remove

Table 14: Data contamination index on KOCO-BENCH, where DCI indicates Data Contamination Index.

Metric	RL		Agent		RAG		Model Optimization		Average	
	Knowledge Corpus	Test Set	Knowledge Corpus	Test Set	Knowledge Corpus	Test Set	Knowledge Corpus	Test Set	Knowledge Corpus	Test Set
DCI	0.037	0.000	0.050	0.020	0.154	0.000	0.080	0.000	0.080	0.005

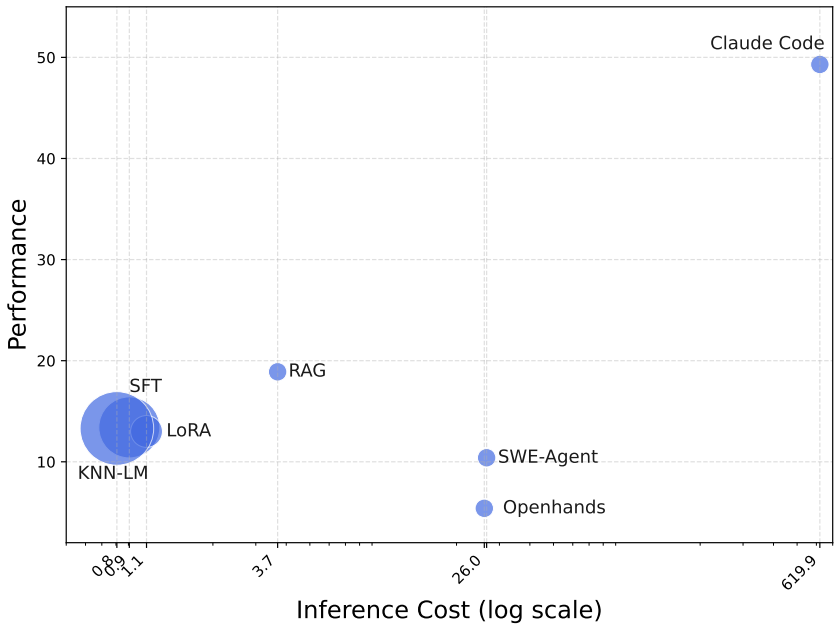

Figure 5: Results of Performance-Cost Trade-off.

the target core functions to prevent data leakage. 2) Training-based methods are trained on the knowledge corpus.

For the domain code generation task, we employ Pass@1 as the primary metric, measuring the proportion of samples where all test cases pass. Since Pass@1 is quite stringent, we also adopt a more lenient metric, AvgPassRate (APR), which measures the average proportion of test cases passed per sample. Additionally, to explore the performance upper bound of models and methods, we use Pass@any, which considers a sample correct if at least one of multiple attempts produces a correct solution. In practice, we generate 10 samples for Pass@any.

Domain Knowledge Understanding. Models are provided with questions and answer options and are asked to select the correct answer(s). Knowledge learning and utilization methods can leverage the knowledge corpus when answering. We use Accuracy (ACC) as the evaluation metric for this task.

By default, we employ greedy decoding (temperature=0) for all evaluations. The only exception is Pass@any, which requires sampling diversity and therefore uses temperature=0.8.

In Table 3, we evaluate 10 well-known LLMs on both Domain Code Generation and Domain Knowledge Understanding tasks, including both closed-source and open-source models across different model families and parameter scales. For closed-source models, we adopt Gemini-2.5-pro (Gemini3), Claude-Sonnet-4.5 (Anthropic, 2025), GPT-5-mini (OpenAI, 2025a) and o4-mini (OpenAI, 2025b). For open-source models, we adopt Kimi-K2-Instruct (moonshotai, 2025), DeepSeek-V3.1 (deepseek-ai, 2025), Qwen2.5-Coder-32B-Instruct (Qwen, 2024a), Qwen2.5-Coder-7B-Instruct (Qwen, 2024b), Llama-3.1-8B-Instruct (meta-llama, 2024) and Deepseek-Coder-7b-Base (deepseek ai, 2025).

K.2 Setup of Domain Specialization Methods

We provide the implementation details of domain specialization methods evaluated in our experiments.

SFT (Dong et al., 2024a) We perform full-parameter fine-tuning based on the Next-Token Prediction (NTP) objective. We use a learning rate of 5×10^{-6} with cosine decay, train for 5 epochs with a maximum sequence length of 2048, and use BF16 precision.

LoRA (Hu et al., 2021) We adopt the PEFT framework, freezing pre-trained weights and capturing domain-specific knowledge through low-rank decomposition matrices. We set the learning rate of 1×10^{-4}, rank $r = 16$, alpha $\alpha = 32$, and dropout to 0.05, training for 5 epochs.

RAG (Lewis et al., 2020) We employ BM25 for lexical retrieval. The knowledge corpus is chunked at both function-level and class-level granularities, covering instance-specific context and framework-wide

knowledge. We retrieve top-5 results using queries constructed from function names and parameter signatures.

kNN-LM (Khandelwal et al., 2020), **kNM-LM** (Tang et al., 2023), and **DSCC** (Yu et al., 2025). We use the official implementations and default configurations provided in the original papers. We adapt these methods to our tasks by modifying the input-output format without altering their core algorithms.

Training Data Organization for SFT and LoRA. Both SFT and LoRA follow the same data organization procedure, consistent with standard practices used in modern code LLMs such as StarCoder and DeepSeek-Coder (deepseek ai, 2025). Specifically, we concatenate all files from the knowledge corpus (raw documentation, source code, examples, etc.) in random order, with special separators `<|endoftext|>` inserted between files. The concatenated text is then split into equal-length training chunks based on the maximum context length. All chunks participate in training, ensuring complete coverage of the knowledge corpus.

K.3 Setup of LLM Agents

We evaluate three agentic coding systems on KoCo-Bench: SWE-Agent (Yang et al., 2024), OpenHands (Wang et al., 2025), and Claude Code (ClaudeCode). SWE-Agent is configured with five exploration tools (`find_file`, `search_dir`, `search_file`, `view_code`, `filemap`). We set max_turns=30 and timeout=3000s per task. OpenHands uses the CodeAct agent with Terminal Tool for bash execution and file operations. We limit execution to max_steps=30 per task. Claude Code is invoked via CLI, allowing `Read/Write/Edit/Grep/Glob/Bash` tools. We set max_turns=30 and timeout=3000s per task, requiring structured JSON output for reliable result extraction.

K.4 Experiment Setup of Investigating Knowledge Forgetting and Conflict

Understanding these is essential for developing effective continual learning strategies in domain-specific software development. We investigate this problem on KoCo-Bench using SFT, which offers a controlled experimental setting to isolate the effects of knowledge accumulation without confounding factors from retrieval or auxiliary inference mechanisms. We compare three knowledge learning scenarios: (1) *Single Framework* (default): train on the knowledge corpus of one framework and evaluate on projects built with that framework, (2) *Multi-Framework, Same Domain*: sequentially train on two frameworks within the same domain (e.g., two RL frameworks), then evaluate on the first-learned framework to measure intra-domain knowledge retention, and (3) *Multi-Framework, Cross-Domain*: sequentially train on two frameworks from distinct domains (e.g., a RL framework followed by a agent framework), then evaluate on the first-learned framework to assess cross-domain interference.

Case 1 (Invalid API Call)

Prompt: This function implements computation and application of token-level KL penalty. It computes token-level KL divergence based on **kl_penalty** type, then subtracts KL penalty term from original rewards to obtain token-level rewards, and computes sequence-level KL divergence and batch-level KL divergence. The function also needs to implement KL controller update.

Qwen2.5-Coder-7B-instruct

```
AttributeError: 'FixedKLController' object has no attribute 'compute_kl_divergence'
```

```python
def apply_kl_penalty(data: DataProto, kl_ctrl: KLController, kl_penalty="kl"):
    # Compute token-level KL divergence based on kl_penalty type
    if kl_penalty == "kl":
        kl_divergence = kl_ctrl.compute_kl_divergence(data.token_level_scores, data.old_log_probs,
                                                      data.ref_log_probs)
    elif kl_penalty == "abs":
        kl_divergence = kl_ctrl.compute_abs_kl_divergence(data.token_level_scores, data.old_log_probs,
                                                          data.ref_log_probs)
    elif kl_penalty == "mse":
        kl_divergence = kl_ctrl.compute_mse_kl_divergence(data.token_level_scores, data.old_log_probs,
                                                          data.ref_log_probs)
    elif kl_penalty == "low_var_kl":
        kl_divergence = kl_ctrl.compute_low_var_kl_divergence(data.token_level_scores, data.old_log_probs,
                                                              data.ref_log_probs)
    elif kl_penalty == "full":
        kl_divergence = kl_ctrl.compute_full_kl_divergence(data.token_level_scores, data.old_log_probs,
                                                           data.ref_log_probs)
    else:
        raise ValueError("Invalid kl_penalty type")

    # Subtract KL penalty term from original rewards to obtain token-level rewards
    data.token_level_rewards = data.token_level_scores - kl_divergence
    # Compute sequence-level KL divergence and batch-level KL divergence
    sequence_kl_divergence = kl_ctrl.compute_sequence_kl_divergence(data.token_level_scores, data.old_log_probs,
                                                                    data.ref_log_probs)
    batch_kl_divergence = kl_ctrl.compute_batch_kl_divergence(data.token_level_scores, data.old_log_probs,
                                                              data.ref_log_probs)
    # Update KL controller
    kl_ctrl.update_kl_controller(sequence_kl_divergence, batch_kl_divergence)
    # Return updated DataProto and metrics
    metrics = {"critic/kl": batch_kl_divergence, "critic/kl_coef": kl_ctrl.kl_coef}
    return data, metrics
```

Ground Truth

```python
def apply_kl_penalty(data: DataProto, kl_ctrl: KLController, kl_penalty="kl"):
    token_level_scores = data.batch["token_level_scores"]
    batch_size = data.batch.batch_size[0]
    response_mask = data.batch["response_mask"]

    # compute kl between ref_policy and current policy
    kld = compute_kl(data.batch["old_log_probs"],
                     data.batch["ref_log_probs"],
                     kl_penalty=kl_penalty)
    kld = kld * response_mask  # (batch_size, response_length)

    data.batch["token_level_rewards"] = token_level_scores - kl_ctrl.kl_coef * kld

    current_kl = VF.masked_mean(kld, mask=response_mask, dim=-1)  # average over sequence
    current_kl = torch.mean(current_kl, dim=0).item()

    metrics = {"critic/kl": current_kl, "critic/kl_coef": kl_ctrl.kl_coef}

    # According to https://github.com/huggingface/trl/blob/v0.11.0/trl/trainer/ppo_trainer.py#L880
    kl_ctrl.update(current_kl=current_kl, n_steps=batch_size)

    return data, metrics
```

Figure 6: A case of invalid API call.

Case 2 (Data Constraint Violations)

Prompt: [Detailed Description] Function supports advantage computation with multi-source reward fusion. Reward sources include reward model scores and accuracy rewards. Groups multiple responses generated from the same prompt by n_samples intervals. Uses RLOO to compute advantage function for each reward source and fuses multiple reward sources, computes cumulative return values after reward fusion and applies whitening.

Qwen2.5-Coder-7B-instruct

```
RuntimeError: stack expects each tensor to be equal size, but got [2, 6] at entry 0 and [2] at entry 1
```

```python
import torch
import verl
def compute_rloo_advantage_return(data: verl.DataProto, response_mask: torch.Tensor, n_samples, config):
    # Extract reward model scores and accuracy labels
    rm_scores = data.batch["rm_scores"]
    acc = data.batch["acc"]
    # Compute DPO reward and accuracy reward
    dpo_reward = config.algorithm.reward_dpo_coef * rm_scores
    acc_reward = config.algorithm.reward_gt_coef * acc
    # Group responses by n_samples intervals
    grouped_rm_scores = torch.split(rm_scores, n_samples, dim=0)
    grouped_acc = torch.split(acc, n_samples, dim=0)
    # Compute RLOO advantage for each reward source
    rm_advantages = []
    acc_advantages = []
    for i in range(len(grouped_rm_scores)):
        rm_advantage = grouped_rm_scores[i] - grouped_rm_scores[i].mean()
        acc_advantage = grouped_acc[i] - grouped_acc[i].mean()
        rm_advantages.append(rm_advantage)
        acc_advantages.append(acc_advantage)
    # Fuse multiple reward sources
    fused_advantages = torch.stack(rm_advantages + acc_advantages, dim=0)
    fused_advantages = fused_advantages.mean(dim=0)
    # Compute cumulative return values
    returns = torch.cumsum(fused_advantages, dim=1)
    # Apply whitening
    advantages = (fused_advantages - fused_advantages.mean()) / fused_advantages.std()
    return advantages, returns
```

Ground Truth

```python
def compute_rloo_advantage_return(data: verl.DataProto, response_mask: torch.Tensor, n_samples, config):
    # calculate rloo reward on different reward sources, and sum again
    def masked_rloo(reward_tensor_original, mask_tensor):
        reward_tensor = reward_tensor_original.clone()
        reward_tensor[~mask_tensor] = 0
        for start_pos in range(0, reward_tensor.shape[0], n_samples):
            cur_rewards_mean = torch.cat(
                [reward_tensor[pos : pos + 1][mask_tensor[pos : pos + 1]].mean(dim=0, keepdim=True)
                    for pos in range(start_pos, start_pos + n_samples)],dim=0)
            cur_rewards_sum = cur_rewards_mean.sum()
            cur_reward_baseline = cur_rewards_sum / (n_samples - 1)
            reward_tensor[start_pos : start_pos + n_samples][mask_tensor[start_pos : start_pos + n_samples]] = (
                reward_tensor[start_pos : start_pos + n_samples][mask_tensor[start_pos : start_pos + n_samples]]
                * (n_samples / (n_samples - 1))- cur_reward_baseline)
        return reward_tensor
    reward_tensors = []
    with torch.no_grad():
        if "rm_scores" in data.batch.keys() and config.algorithm.reward_dpo_coef != 0.0:
            reward_tensor = data.batch["rm_scores"]
            reward_mask = response_mask.bool()
            reward_tensors.append(masked_rloo(reward_tensor, reward_mask) * config.algorithm.reward_dpo_coef)
        if "acc" in data.batch.keys() and config.algorithm.reward_gt_coef != 0.0:
            reward_tensor = torch.zeros_like(response_mask, dtype=torch.float32)
            reward_mask = torch.zeros_like(response_mask, dtype=torch.bool)
            prompt_ids = data.batch["prompts"]
            prompt_length = prompt_ids.shape[-1]
            valid_response_length = data.batch["attention_mask"][:, prompt_length:].sum(-1)
            reward_mask[torch.arange(0, valid_response_length.shape[0], dtype=torch.long,
                                device=valid_response_length.device), valid_response_length - 1] = True
            reward_tensor[torch.arange(0, valid_response_length.shape[0], dtype=torch.long,
                                device=valid_response_length.device),valid_response_length - 1] = data.batch["acc"]
            reward_tensors.append(masked_rloo(reward_tensor, reward_mask) * config.algorithm.reward_gt_coef)
        final_reward_tensor = sum(reward_tensors)
        returns = (final_reward_tensor * response_mask).flip(dims=[-1]).cumsum(dim=-1).flip(dims=[-1])
        advantages = returns.clone()
        advantages = verl_F.masked_whiten(advantages, response_mask)
        return advantages, returns
```

Figure 7: A case of data constraint violations.