

# MapCoder-Lite: Distilling Multi-Agent Coding into a Single Small LLM

Woongkyu Lee<sup>1</sup> Junhee Cho<sup>2</sup> Jungwook Choi<sup>1\*</sup>

<sup>1</sup>Hanyang University <sup>2</sup>Samsung SDS  
{lwghanyang, choij}@hanyang.ac.kr  
junhee.cho@samsung.com

## Abstract

Large language models (LLMs) have advanced code generation from single-function tasks to *competitive-programming* problems, but existing multi-agent solutions either rely on costly large-scale ( $> 30B$ ) models or collapse when downsized to small open-source models. We present *MapCoder-Lite*, a framework for distilling the complex reasoning of large, multi-agent coding systems into a single 7B model. Our contribution is a novel, three-pillar methodology that synergistically generates, refines, and encodes multi-agent knowledge: (i) *pass-based trajectory distillation* from strong LLMs fixes format fragility in retrieval and reduces failures in debugging, (ii) *supervisor-guided correction* with global feedback strengthens planning and coding agents, and (iii) *agent-wise LoRA fine-tuning* delivers memory-efficient specialisation. Comprehensive evaluation on xCodeEval, APPS, and CodeContests shows that MapCoder-Lite more than doubles xCodeEval accuracy (13.2%  $\rightarrow$  28.3%), eliminates all format failures, while reducing GPU memory and token-generation time by  $4\times$  compared to a 32B model. It also achieves over 10% gains on simpler coding benchmarks, demonstrating broad improvements beyond competitive programming. These results demonstrate that careful agent-wise fine-tuning unleashes high-quality multi-agent coding on a small language model. Our code is publicly available at <https://github.com/aiha-lab/MapCoder-Lite>.

## 1 Introduction

LLMs have revolutionized code synthesis, achieving near-perfect accuracy on function-level tasks like HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021). Research has now shifted to *competitive programming*, which demands efficient algorithms, robust implementation, and resilience against hidden test cases. Benchmarks like CodeElo (Quan et al., 2025) highlight

significant challenges that have spurred the development of massive models, such as OpenAI’s o3 (OpenAI et al., 2025) and the 671B-parameter DeepSeek-V3 (DeepSeek-AI et al., 2025b). These models report strong performance on Codeforces and related benchmarks despite their computational costs and proprietary nature.

Competitive programming involves algorithmically complex problem solving under strict constraints, making it a difficult setting for language models. Single-agent prompting, where one model handles the entire problem-solving process, often falls short (Wei et al., 2023; Jiang et al., 2024b; Yasunaga et al., 2024). To overcome this, recent work has explored *multi-agent code-generation* frameworks that split the task into stages and assign each to a dedicated agent, improving end-to-end performance through role specialization (Huang et al., 2024; Hong et al., 2024; Islam et al., 2024, 2025). MapCoder (Islam et al., 2024) (Fig. 1) exemplifies this approach, coordinating specialized agents throughout the pipeline.

The effectiveness of multi-agent frameworks typically relies on large-scale ( $> 30B$ ) open models (OpenAI et al., 2024; Comanici et al., 2025), which possess the capacity to perform a wide range of specialized roles required across the multi-agent pipeline. Due to their multi-step nature, these frameworks incur significantly higher token usage and API calls than single-agent setups, resulting in increased latency and computational cost.

Their impracticality in resource-constrained settings naturally motivates small-model multi-agent solutions, which align well with the growing industry trend toward on-device AI (Gunter et al., 2024; Zhang et al., 2023). In practice, however, deploying small language models (SLMs) under 10B parameters—even those with strong coding abilities (Qwen et al., 2025)—often yields limited accuracy gains compared to single-model direct prompting. This performance drop stems from

\*Corresponding author.

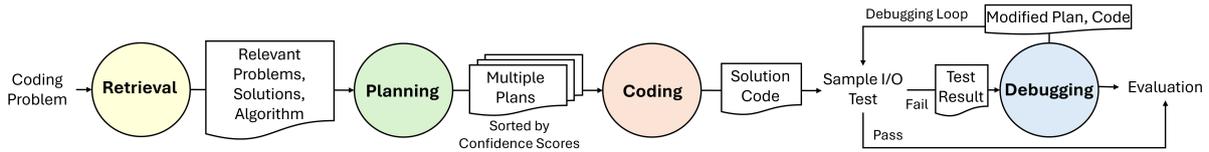


Figure 1: Overview of the MapCoder system. Given a natural language problem, the retrieval agent fetches relevant algorithmic knowledge, followed by the planning agent generating a solution plan. The coding agent implements the plan, and the debugging agent iteratively refines the code based on test outcomes.

SLMs’ difficulty in following the structured formats (e.g., XML) required for multi-agent communication (Xia et al., 2024) and their limited capacity to support the complex reasoning needed across agent roles, leading to failures in retrieval, planning, or debugging.

To make SLM-based multi-agent systems effective, fine-tuning becomes a necessary step. However, this approach poses three key obstacles. First, existing code datasets do not provide intermediate artefacts aligned with the roles in a multi-agent pipeline. Second, fine-tuning each agent independently fails to account for inter-agent dependencies. As illustrated in Fig. 1, later stages rely on the outputs of earlier agents, so errors in one stage can propagate and mislead downstream components. Third, training separate models for each agent increases GPU memory consumption at inference time, undermining the efficiency benefits of using small models in the first place.

We address these challenges with *MapCoder-Lite*, the first multi-agent coding framework that drives a single 7B backbone—extended only by lightweight, role-specific adapters—to performance near that of 32B systems. *MapCoder-Lite* is built on three components:

- **Pass-based trajectory distillation from strong LLMs (Sect. 5.1):** To address the lack of role-specific training data, we collect trajectories from strong LLMs. However, fine-tuning on outputs that are only locally valid often fails to yield correct final solutions. We overcome this by implementing *pass-based filtering* that exclusively retains trajectories whose final code passes all unit tests.
- **Supervisor-aided cross-agent refinement (Sect. 5.2):** We employ a *supervisor* model that analyzes the full trajectory generated by the small model to detect cross-agent failure patterns and regenerate the responsible agent’s output, guiding the model toward global suc-

cess and helping bridge the capacity gap.

- **Memory-efficient LoRA specialization (Sect. 5.3):** We show that in complex multi-agent settings, LoRA (Hu et al., 2021) achieves better accuracy and efficiency than full fine-tuning. This enables all agents to share a frozen Qwen2.5-7B backbone with lightweight, role-specific adapters, adding under 3% extra parameters.

We conducted a comprehensive evaluation on three representative competitive-programming suites—xCodeEval (Khan et al., 2024), APPS (Hendrycks et al., 2021), and CodeContests (Li et al., 2022)—and found that *MapCoder-Lite* leverages trajectory distillation, supervisor-guided cross-agent refinement, and rank-32 LoRA specialisation to boost xCodeEval accuracy from 13.2% to 28.3%, eliminates every XML-schema failure, all while cutting GPU memory and token generation time (time per output token) by 4×. These results demonstrate that careful agent-wise fine-tuning can unlock high-quality multi-agent code generation on small language models.

## 2 Related Work

### 2.1 LLMs for Competitive Programming

LLMs have shown strong performance on *function-level* code generation tasks (Hui et al., 2024; Guo et al., 2024; Rozière et al., 2024), achieving near-perfect scores on HumanEval (Chen et al., 2021) and MBPP (Austin et al., 2021). Recent work shifts to the harder setting of *competitive programming*, which requires generating full, efficient programs that pass hidden tests. Benchmarks like APPS (Hendrycks et al., 2021), CodeContests (Li et al., 2022), and xCodeEval (Khan et al., 2024) capture this challenge and now define the state of the art (OpenAI et al., 2025; DeepSeek-AI et al., 2025a), motivating the multi-agent systems that follow.

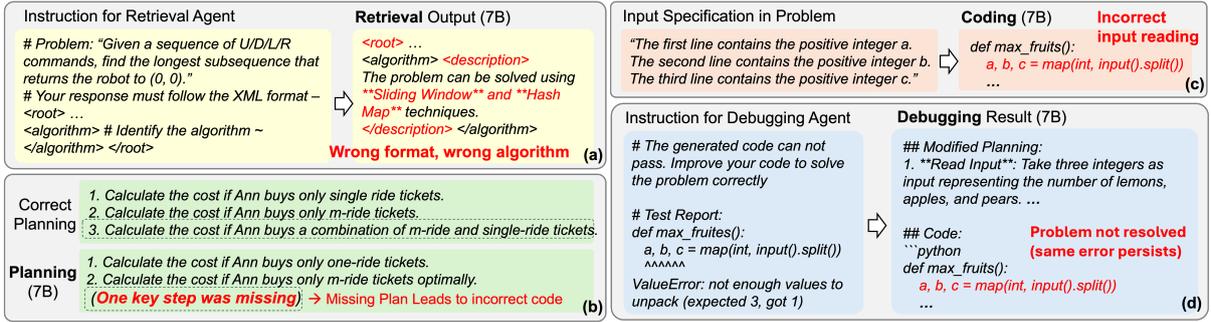


Figure 2: Representative failure cases of the 7B-scale model across all agents. (a) Retrieval: invalid XML format and incorrect algorithm. (b) Planning: missing key step. (c) Coding: misinterpreted input specification. (d) Debugging: persistent unresolved error. Detailed illustrations are provided in Appendix E.

## 2.2 Multi-Agent Code Generation

Recent work has proposed *multi-agent pipelines* to better handle the linguistic complexity and algorithmic subtlety of competitive programming, which often cause single prompts to miss edge cases or violate constraints (Huang et al., 2024; Pan et al., 2025). Among these, *MapCoder* (Islam et al., 2024) adopts a four-stage pipeline (Fig. 1) that performs well on APPS and CodeContests. A *retrieval* agent first selects an algorithm from a private corpus and returns a schema-constrained XML snippet. A *planning* agent then expands this into step-wise plans with confidence scores. The top plan is passed to a *coding* agent, which generates the full source code. Finally, a *debugging* agent tests and patches the code until it passes all unit tests, backtracking to alternative plans if needed. This modular design improves accuracy via role specialization, but also demands broad capability from the underlying LLM across all stages.

## 2.3 Task-Specific Fine-Tuning

MapCoder uses strong LLMs at each stage, trading efficiency for accuracy. A natural question is whether a single small model (<10B), fine-tuned per role, can achieve similar results. While prior work has shown the value of fine-tuning in multi-agent (Zhao et al., 2025; Liang et al., 2025; Shen et al., 2024) and code tasks (Fan et al., 2025; Tsai et al., 2024; Yu et al., 2024; Jiang et al., 2024a), no study has systematically explored role-aligned fine-tuning for small LLMs in competitive programming pipelines. To our knowledge, however, no work has comprehensively studied how far role-aligned fine-tuning can push a small LLM inside a multi-agent pipeline for competitive programming.

## 3 Analysis of Multi-Agent Limitations

### 3.1 High Cost with Large Models

We evaluated MapCoder using the Qwen2.5-32B-Instruct model on the CodeContests benchmark, which comprises 165 competitive programming problems. The system required 27.53 hours of runtime, processed approximately 5.08 million input tokens and 1.60 million output tokens, and made 3,095 API calls. This substantial resource usage stems from MapCoder’s multi-agent design involving four distinct agents and multiple iterations for planning and debugging. These results highlight the heavy runtime and memory burden of using large-scale language models throughout the pipeline. We therefore hypothesize that replacing each agent with a fine-tuned SLM can substantially reduce token-generation time and GPU memory usage, even under comparable token and API call counts.

### 3.2 Failure Cases of Small Models

**Format Following Failures.** Multi-agent systems often rely on structured outputs in predefined formats (Yang et al., 2025; Tang et al., 2025). In MapCoder, for example, retrieval and planning agents are required to produce XML-formatted responses such as <root>, <algorithm>, and <confidence> for downstream parsing. However, small models frequently violate these schemas—omitting or misplacing tags—which disrupts subsequent processing and halts pipeline execution (Fig. 2a). This issue is amplified by the weaker format-following ability of open-source SLMs compared to proprietary LLMs (Xia et al., 2024), underscoring the importance of strict structural adherence in multi-agent workflows.

**Low Role Performance.** Small models often

struggle with role-specific tasks due to limited capacity. Fig. 2 illustrates representative failures across agents: (a) the retrieval agent produces incorrect or misleading algorithm descriptions, corrupting shared context for all downstream stages; (b) the planning agent outputs superficially correct but incomplete plans, often missing subtle edge cases; (c) the coding agent introduces logical or I/O errors even when given valid plans, yielding incorrect or unexecutable code; and (d) the debugging agent fails to detect or fix simple bugs, resulting in repeated ineffective patches. Together, these failures indicate that, without stronger supervision or greater capacity, individual agents act as bottlenecks that undermine the reliability of the entire pipeline.

## 4 Challenges

The multi-agent approach provides a parameter-free mechanism for orchestrating specialized roles via prompting, yet SLMs consistently underperform in this zero-shot setting. This suggests that prompt-based assignment alone is insufficient for SLMs, necessitating a shift toward parameter-driven optimization (Du et al., 2025). However, effectively fine-tuning SLMs within a multi-agent framework presents several non-trivial challenges.

### 4.1 Lack of Role-Specific Training Data.

Multi-agent fine-tuning requires high-quality intermediate supervision tailored to each agent’s role. Existing approaches rely either on reconstructing intermediate signals from public datasets (Shen et al., 2024) or on self-collection using SLM-generated outputs (Liang et al., 2025). However, public code benchmarks are not designed for multi-agent pipelines and lack role-aligned artefacts, while self-collected data from small models is often malformed or incomplete due to limited capacity and task complexity. This underscores the need to generate new, role-specific supervision tailored to multi-agent training.

### 4.2 Limited Global Awareness.

In multi-agent workflows, information flows sequentially, making end-to-end success reliant on the coherence of intermediate outputs across stages. Yet agents trained in isolation lack awareness of these dependencies, leading to local errors that propagate and compromise the final outcome. For instance, when the final code fails, it is often unclear whether the root cause lies in a flawed plan

from the planning agent or in an incorrect implementation by the coding agent, even if each agent appears to perform its role plausibly. Such failures show that success depends not only on individual competence but also on aligned interactions. Without global awareness, even well-tuned agents may fall short.

### 4.3 Inefficiency of Full Model Fine-Tuning

Previous multi-agent fine-tuning approaches typically rely on OpenAI’s fine-tuning API (Zhao et al., 2025) or full-parameter tuning for each agent (Shen et al., 2024; Zeng et al., 2023). However, full fine-tuning scales memory usage linearly with the number of agents, negating one of the primary advantages of SLMs—their efficiency in memory and deployment. Furthermore, full fine-tuning does not necessarily guarantee superior performance over parameter-efficient methods such as LoRA (Hu et al., 2021).

## 5 Methodology

To overcome the limitations of applying multi-agent code generation to SLMs, we propose a *role-aligned supervised fine-tuning* pipeline that equips a single 7B backbone with specialized behavior for four agents via lightweight LoRA adapters. This design directly addresses the three main challenges identified in our analysis:

- **Lack of role-specific data.** To compensate for the absence of intermediate supervision, we distill high-quality trajectories from strong LLMs, using execution results to retain only clean, agent-specific samples (Sec. 5.1).
- **Limited global awareness.** To maintain cross-agent consistency, we introduce a *supervisor-guided refinement* mechanism that detects failures and regenerates only the faulty component, yielding coherent, context-aware training data (Sec. 5.2).
- **Inefficiency of full-model fine-tuning.** We address parameter and memory overhead by applying rank-32 LoRA adapters to a shared frozen 7B backbone, enabling agent-wise specialization with minimal additional cost (Sec. 5.3).

Together, these techniques allow us to retain the advantages of the multi-agent approach while making it viable for deployment on open-source, resource-efficient language models.

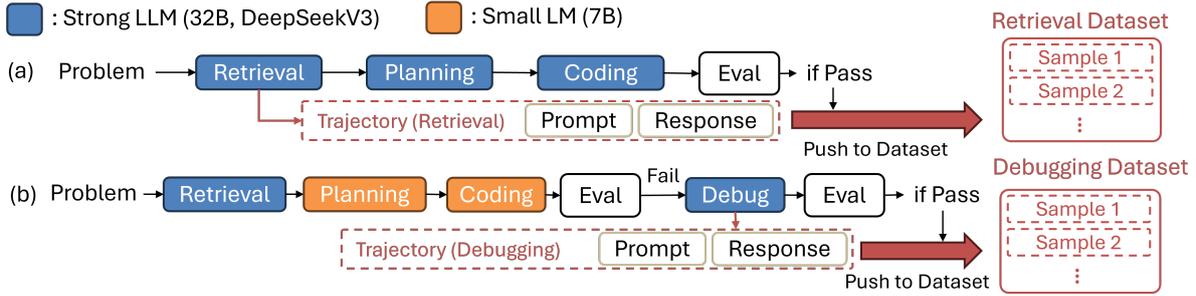


Figure 3: Illustration of trajectory construction for retrieval and debugging datasets. (a) When a strong LLM generates a complete solution that passes unit tests, the retrieval agent’s input-output pair is extracted as a training example. (b) To collect debugging data, a 7B model is used for planning and coding, and the strong LLM is used for debugging when the initial code fails. If the revised output passes, the debugging trajectory is added to the dataset.

Retrieval		Planning / Coding		xCodeEval
Dataset	Filtering	Dataset	Filtering	Accuracy (%)
-	-	-	-	11.32
Strong	-	-	-	16.04
Strong	Format	-	-	16.98
Strong	Pass	Strong	Pass	18.87
<b>Strong</b>	<b>Pass</b>	<b>Supervisor</b>	<b>Pass</b>	<b>22.64</b>

Table 1: Ablation study on data source and filtering methods for training retrieval, planning, and coding agents on xCodeEval. “Format” filtering keeps samples with valid single-agent outputs, while “Pass” retains only those where the final program passes all unit tests. “Strong” denotes data from a large LLM; “Supervisor” indicates trajectories refined after failure analysis.

## 5.1 Strong LLM for Retrieval and Debugging

Fig. 3 shows the proposed trajectory construction method. Our data pipeline begins by asking *strong LLMs*, namely Qwen2.5-32B-Instruct and DeepSeek-V3, to solve each coding task while explicitly printing intermediate artefacts (i.e., *trajectories*) produced by each MapCoder role.

To build reliable training data, we collect trajectories from strong LLMs and keep only those whose final code passes all unit tests. Unlike rejection sampling based solely on local validity at the single-agent level (Zelikman et al., 2022; Zeng et al., 2023), our *pass-based filtering* ensures end-to-end success across all roles. This focuses fine-tuning on trajectories verified through full execution, yielding accuracy gains over locally valid samples (Table 1, row 2 vs. row 3).

Our analysis further shows that self-collected trajectories from a 7B model often mislabel tasks, such as overpredicting dynamic programming, and exhibit limited algorithmic diversity. As a result, fine-tuning on these traces leads to lower perfor-

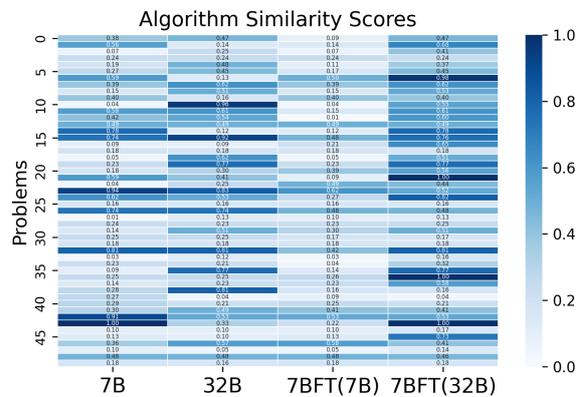


Figure 4: Cosine similarity between algorithm descriptions generated by base and fine-tuned models and ground-truth algorithm tags for 50 xCodeEval problems. Darker cells indicate stronger alignment.

mance (7BFT). In contrast, trajectories generated by 32B align more closely with ground-truth tags and provide more accurate algorithm descriptions. Fine-tuning on these trajectories (7BFT(32B)) significantly improves performance, demonstrating that strong-LLM supervision transfers both correctness and algorithmic diversity to smaller models.

## 5.2 Supervisor-Guided Planning and Coding

Even when fine-tuned on strong-model trajectories, 7B agents struggle to achieve robust multi-stage reasoning. One challenge is the lack of *global awareness*, where agents trained in isolation fail to account for downstream dependencies, an issue previously discussed in Section 4. Another is the *capacity gap* (Bansal et al., 2024; Xu et al., 2025): while strong LLMs produce coherent and mostly correct outputs, small models tend to mimic surface forms without learning the underlying reasoning. These limitations underscore the need for super-

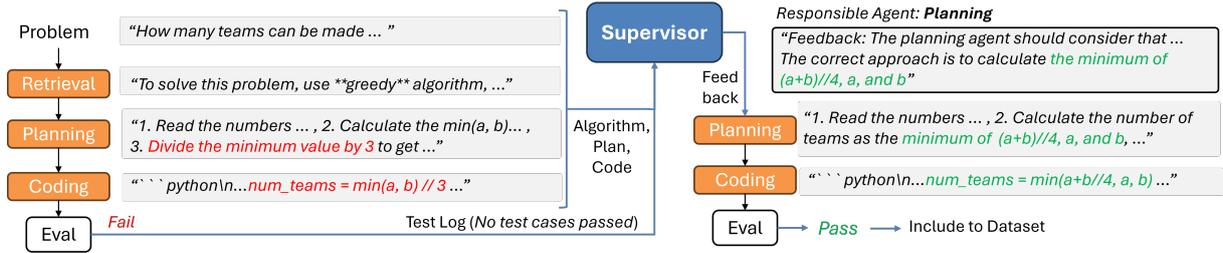


Figure 5: Supervisor-aided data collection pipeline. When the final output fails, the supervisor inspects the full trajectory (including algorithm, plan, code, and test result), identifies the responsible agent, and provides targeted feedback to revise its output. If the revised result passes, the updated trajectory is added to the fine-tuning dataset.

vision that supports both global coordination and deeper abstraction.

To address these issues, we propose a *supervisor-guided refinement pipeline* that supplies global feedback without enlarging the runtime model. As shown in Fig. 5, a 7B MapCoder first produces a complete retrieval→planning→coding trajectory. If the resulting program fails its unit tests, the entire trajectory is forwarded to a high-capacity *supervisor* LLM (DeepSeek-V3). The supervisor analyzes the trace, identifies the agent primarily responsible for the failure, and issues concise, role-specific feedback. Only the selected agent then regenerates its output. The revised trajectory is re-tested; once all tests pass, the final plan–code pair is added to the fine-tuning corpus.

Supervisor-guided refinement is applied selectively to trajectories where the strong LLM succeeds but the 7B model fails, minimizing generation overhead. The supervisor operates *only* during data generation, keeping inference lightweight. We store only the corrected agent input–output pairs, omitting the feedback itself, so the 7B model learns solely from information available at runtime. Crucially, every example in this corpus is produced and execution-validated by the 7B model, eliminating concerns about a capacity gap. Iterating over thousands of problems yields a dataset that better aligns planners and coders with end-to-end success. Table 1 confirms that this strategy delivers a substantial accuracy boost.

### 5.3 Multi-Agent LoRA Fine-Tuning

To enable role-specific specialization, we adopt an *agent-wise LoRA* strategy: all four agents share a frozen 7B backbone, each with independent rank-32 LoRA adapters fine-tuned on role-specific data. Unlike prior approaches that fine-tune the entire model across tasks (Shen et al., 2024; Zeng et al., 2023), our method isolates agent behavior at the

FT Method	Trainable Parameters	Required Memory for Training (GB)	Accuracy (%)
FFT	7615.62M	45.69	18.87
LoRA	20.19M	15.35	22.64

Table 2: The number of trainable parameters, required memory for training, and xCodeEval accuracy of full fine-tuning (FFT) and LoRA methods with MapCoder-Lite 7B. Accuracy is measured only up to the coding stage, without debugging.

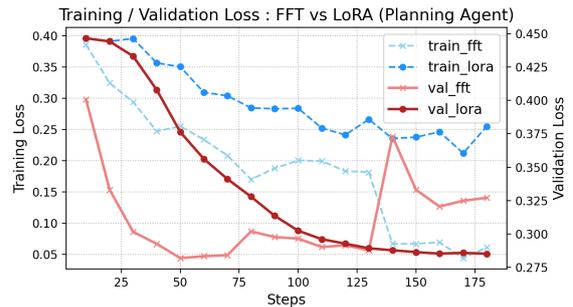


Figure 6: Training and validation loss for the Planning agent using full fine-tuning (FFT) and LoRA. LoRA shows higher training loss but maintains lower and more stable validation loss, suggesting better generalization.

adapter level while preserving a unified backbone.

We empirically show that LoRA serves effectively as a modular control layer in multi-agent code generation, delivering both accuracy and efficiency gains (Table 2). We attribute this to LoRA’s implicit regularization (Hu et al., 2021; Houlsby et al., 2019), which promotes task-specific behavior without overwriting pretrained knowledge. In multi-agent settings with complex, structured I/O, full fine-tuning often overfits to superficial patterns, weakening general reasoning. As shown in Fig. 6, full fine-tuning achieves lower training loss but increasing validation loss, whereas LoRA maintains lower and more stable validation loss, indicating better generalization and adaptability across roles.

Method	xCodeEval (106)			APPS (150)			CodeContests (165)		
	Accuracy↑	Pass Count↑	Format Fails↓	Accuracy↑	Pass Count↑	Format Fails↓	Accuracy↑	Pass Count↑	Format Fails↓
Direct	18.87	20	–	4.00	6	–	1.21	2	–
CoT	5.66	6	–	4.67	7	–	3.64	6	–
Self-planning	9.43	10	–	1.33	2	–	3.03	5	–
Analogical	12.26	13	–	3.33	5	–	1.82	3	–
MapCoder	13.21	14	29	6.00	9	30	6.06	10	59
MapCoder-Lite	<b>28.30</b>	<b>29</b>	<b>0</b>	<b>8.00</b>	<b>12</b>	<b>0</b>	<b>13.33</b>	<b>22</b>	<b>0</b>

Table 3: Performance comparison of different methods using Qwen7B on xCodeEval, APPS, and CodeContests with Pass@1 accuracy (%). The numbers in parentheses indicate the number of problems in each benchmark.

## 6 Experimental Settings

**Models.** All agent-wise fine-tuning experiments use Qwen2.5-7B-Instruct as the base model. For comparison, we include larger variants—Qwen2.5-14B-Instruct and Qwen2.5-32B-Instruct—which serve as upper bounds for evaluating the effectiveness of our lightweight strategy. To assess whether a code-pretrained backbone improves performance, we additionally test Qwen2.5-Coder-7B-Instruct. We also apply our method to Llama3.1-8B and CodeLlama-7B to demonstrate generalizability beyond the Qwen family.

**Benchmarks.** We evaluate models on three competitive programming benchmarks: xCodeEval (Khan et al., 2024), APPS (Hendrycks et al., 2021), and CodeContests (Li et al., 2022). For xCodeEval, we use the 106 problems from the compact program synthesis split. For CodeContests, we evaluate all 165 official test problems. For APPS, we randomly select 150 test-set problems, following MapCoder (Islam et al., 2024). Each benchmark provides natural language problem descriptions with sample I/O and is evaluated using hidden unit tests. We additionally report results on HumanEval and MBPP to assess generalization to function-level tasks.

**Fine-Tuning Setup.** Each agent is fine-tuned using LoRA adapters with rank-32, applied to the linear layers for query, key, value, and output projections. We use a learning rate of  $2e-5$ , gradient accumulation of 16, and train for three epochs, with hyperparameters tuned per agent to ensure stability. All training runs are conducted on a single NVIDIA A100 80GB GPU. The cost of data curation prior to fine-tuning is summarized in Table 8 (Section 7.5).

**Evaluation Setting.** We report pass@1 accuracy using greedy decoding to ensure reproducibility and align with baseline evaluations.

## 7 Results

### 7.1 Effectiveness of Agent-wise Fine-Tuning

Table 3 compares the performance of Qwen2.5-7B under various prompting strategies—direct prompting, CoT(Wei et al., 2023), self-planning(Jiang et al., 2024b), analogical (Yasunaga et al., 2024)—and multi-agent setups. Without fine-tuning, MapCoder yields only marginal improvements and even underperforms direct prompting on xCodeEval (13.21% vs. 18.87%), underscoring the difficulty of coordinating small models in multi-agent pipelines due to format errors and poor inter-agent coordination.

In contrast, MapCoder-Lite with agent-wise fine-tuning achieves substantial improvements across all benchmarks. On xCodeEval, accuracy more than doubles to 28.30%, outperforming all prompting baselines. Similar gains are observed on APPS (6.00%  $\rightarrow$  8.00%) and CodeContests (6.06%  $\rightarrow$  13.33%). Notably, fine-tuning eliminates all format failures, indicating that even small LMs can produce structurally consistent outputs when role-specialized. These results highlight the critical role of agent-wise fine-tuning in realizing the full potential of small LMs in multi-agent frameworks, with qualitative examples in Appendix E further illustrating improvements across all roles.

### 7.2 Comparison with Larger Models

Table 4 compares MapCoder’s performance across backbone sizes from 7B to 32B (Qwen[7–32]B), alongside MapCoder-Lite—our fine-tuned 7B variant (Qwen7B-FT). Qwen7B-FT delivers a substantial accuracy boost over the untuned Qwen7B, matching the performance of Qwen14B and coming within six points of Qwen32B. This is particularly notable given that Qwen7B-FT requires only one-quarter of the GPU memory of Qwen32B, making it viable for deployment on memory-constrained edge devices (Karumbunathan, 2022). Moreover, the reduced model size yields a propor-

Model	xCodeEval	APPS	CodeContests	Memory	TPOT
Qwen7B	13.21	6.00	6.06	15.26	12.29
<b>Qwen7B-FT</b>	<b>28.30</b>	<b>8.00</b>	<b>13.33</b>	<b>15.64</b>	<b>12.29</b>
Qwen14B	28.30	10.00	15.76	29.58	23.36
Qwen32B	33.02	13.33	18.18	65.56	45.06

Table 4: Pass@1 accuracy (%) of MapCoder using Qwen models of different scales on xCodeEval, APPS, and CodeContests. Memory usage (GB) and Time Per Output Tokens (ms).

Agent				xCodeEval	Pass	Pass	Format
R	P	C	D	Accuracy (%)	w/o Debug $\uparrow$	w/ Debug $\uparrow$	Fail $\downarrow$
-	-	-	-	13.21	11	3	29
✓	-	-	-	18.87	17	3	12
✓	✓	-	-	26.42	18	10	0
✓	✓	✓	-	24.53	24	2	0
✓	✓	-	✓	27.36	17	12	0
✓	✓	✓	✓	<b>28.30</b>	<b>24</b>	<b>6</b>	<b>0</b>

Table 5: Ablation study of per-agent fine-tuning contributions in MapCoder on xCodeEval. R, P, C, and D denote Retrieval, Planning, Coding, and Debugging agents, respectively.

tional speedup in token generation—LLM decoding being memory-bound—achieving a 4 $\times$  reduction in time-per-output-token (TPOT), as measured using vLLM (Kwon et al., 2023). These results highlight that targeted fine-tuning enables small models to achieve competitive performance with dramatically lower cost.

### 7.3 Ablation Study

#### Contribution of Individual Agent Fine-Tuning.

Table 5 presents an ablation study of agent-wise fine-tuning on xCodeEval. Tuning the retrieval agent alone raises accuracy from 13.21% to 18.87% and cuts format failures from 29 to 12. Adding the planning agent further improves accuracy to 26.42% and eliminates all format errors, boosting both non-debug and debug-assisted completions. Fine-tuning the coding agent (without debugging) increases non-debug passes to 24 but reduces debug-assisted ones, as stronger initial code often bypasses the debugger. In contrast, tuning the debugging agent (without coding) improves recovery from weak code, achieving 27.36% accuracy and 12 debug-assisted completions. Full fine-tuning of all agents yields the best performance (28.30%), confirming that each agent contributes uniquely and coordinated tuning is critical for optimal results.

**Impact of LoRA Rank.** We study the effect of LoRA rank on multi-agent performance to justify the configuration used in this work. The rank-32

Retrieval	Planning	Coding	xCodeEval (%)
32	32	32	<b>22.64</b>
8	32	32	18.87
32	8	32	20.75
32	32	8	21.70
8	8	8	21.70
64	32	32	16.98
32	64	32	20.75
32	32	64	20.75
64	64	64	16.98

Table 6: Effect of LoRA rank on multi-agent performance. The three leftmost columns indicate the LoRA ranks applied to the retrieval, planning, and coding agents, respectively.

setting was selected based on empirical evaluation rather than by convention. As shown in Table 6, rank-32 achieves the highest xCodeEval accuracy (22.64%) among all tested configurations. Lower-rank settings (e.g., (8, 8, 8)) underperform despite being parameter-efficient, while higher-rank configurations (e.g., 64) do not yield further gains and can degrade performance. These results suggest that multi-agent performance depends on interactions between agent roles rather than parameter count alone. Overall, rank-32 offers the best balance between accuracy and efficiency, introducing only about 3% additional trainable parameters relative to the backbone.

**Backbone Selection.** Choosing the right backbone is critical for multi-agent code generation, where each agent performs a distinct, reasoning-intensive role. Among SLM candidates, we compare a general-purpose model (Qwen2.5-7B-Instruct (Qwen et al., 2025)) and a code-specialized variant (Qwen2.5-Coder-7B-Instruct (Hui et al., 2024)) for use in MapCoder. Despite similar sizes, the general-purpose model achieves higher accuracy and format adherence. Agent-wise ablations further show that replacing even one agent with its coder counterpart degrades performance, suggesting that the coder model lacks the contextual reasoning and adaptability needed for role-specific tasks. These results support our choice of the general-purpose model as the unified backbone for MapCoder-Lite (see Appendix B).

### 7.4 Generalization and Robustness

**Evaluation on Simpler Coding Tasks.** To evaluate generalization beyond competition-level programming, we compare MapCoder and MapCoder-Lite on two function-level benchmarks: HumanEval and MBPP. As shown in Table 7,

Method	HumanEval (164 problems)				MBPP (397 problems)			
	Accuracy (%)	Pass w/o Debug $\uparrow$	Pass w/ Debug $\uparrow$	Format Fails $\downarrow$	Accuracy (%)	Pass w/o Debug $\uparrow$	Pass w/ Debug $\uparrow$	Format Fails $\downarrow$
MapCoder	70.73	105	11	14	67.51	244	24	68
MapCoder-Lite	82.93	120	16	1	84.63	305	31	0

Table 7: Evaluation of MapCoder and MapCoder-Lite on HumanEval and MBPP benchmarks.

MapCoder-Lite outperforms the baseline by significantly reducing format-related failures and achieving higher end-to-end accuracy. It improves both coding (increased passes without debugging) and debugging (even greater gains in passes with debugging), indicating enhanced reliability. These results suggest that MapCoder-Lite not only preserves generalization capability but also improves robustness on structurally simpler tasks.

**Model Compatibility.** Our method is architecture-agnostic and applicable to LoRA-supported backbones. Beyond the Qwen2.5 family, MapCoder-Lite improves accuracy and reduces format failures on Llama3.1-8B and CodeLlama-7B across xCodeEval and HumanEval (Appendix D). We also evaluate MapCoder-Lite on the recent reasoning-oriented model Qwen3-4B; despite strong single-agent prompting performance, stable multi-agent behavior is only achieved after agent-wise fine-tuning, with detailed results reported in Appendix C.

### 7.5 Data Curation Cost and Statistics

Table 8 summarizes the dataset size, resource usage, and data generation time for each agent. In particular, we curate approximately 2.3k execution-verified trajectories for the retrieval agent, 1.1k each for the planning and coding agents, and 4.2k for the debugging agent. We used  $2 \times A100$  GPUs with vLLM serving to run Qwen2.5-32B and Qwen2.5-7B models, and partially relied on the DeepSeek API as supervisor and debugging agent, incurring \$20–30. Data generation is performed only once, and the resulting datasets are reusable across models and experiments. The strong supervisor model is invoked only in failure cases to provide brief corrective feedback, rather than full trajectory generation, keeping strong-model involvement infrequent and tightly bounded to training-time supervision. Retrieval and debugging took longer due to longer outputs; debugging in particular depends on planning and coding outputs, requiring upstream completion. While data was generated per agent during development, the pipeline can be streamlined into a single pass for faster future use.

	Retrieval	Planning, Coding	Debugging
Dataset size	2.3k	1.1k (each)	4.2k
Time	2–3 days	16 hours	4–5 days
GPUs	$2 \times A100$	$2 \times A100$	$2 \times A100$

Table 8: Dataset size, data generation time, and computational resource usage for data curation for each agent.

## 8 Conclusion

We introduced *MapCoder-Lite*, a multi-agent coding framework built on a 7B model with lightweight LoRA adapters. Through trajectory distillation, supervisor-guided refinement, and agent-wise specialization, it achieves competitive performance at substantially lower cost. Our results demonstrate that small language models, when carefully fine-tuned, can match the reliability and accuracy of much larger systems in multi-agent code generation.

## Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2025-00561961). This work was also supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2025-02214497, Development of Low-Level Optimization Program API Technology for AI Semiconductors).

## Limitations

Our framework relies on distilled trajectories from strong models such as DeepSeek-V3 and Qwen2.5-32B, yet the fine-tuned 7B model does not fully replicate their performance. Rather than indicating a hard performance ceiling, this gap highlights opportunities to further enhance small models, potentially through architectural extensions or improved training objectives. In addition, the multi-agent structure introduces a broad design space for fine-tuning, as each agent has a distinct role and optimization objective. While we adopt a uniform tuning recipe across agents in this work, more adaptive tuning could further improve performance.

## References

- 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.
- Hritik Bansal, Arian Hosseini, Rishabh Agarwal, Vinh Q. Tran, and Mehran Kazemi. 2024. [Smaller, weaker, yet better: Training llm reasoners via compute-optimal sampling](#). *Preprint*, arXiv:2408.16737.
- 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.
- Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, Luke Marris, Sam Petulla, Colin Gaffney, Asaf Aharoni, Nathan Lintz, Tiago Cardal Pais, Henrik Jacobsson, Idan Szpektor, Nan-Jiang Jiang, and 3290 others. 2025. [Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities](#). *Preprint*, arXiv:2507.06261.
- DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, and 181 others. 2025a. [Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning](#). *Preprint*, arXiv:2501.12948.
- DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, and 181 others. 2025b. [Deepseek-v3 technical report](#). *Preprint*, arXiv:2412.19437.
- Shangheng Du, Jiabao Zhao, Jinxin Shi, Zhentao Xie, Xin Jiang, Yanhong Bai, and Liang He. 2025. [A survey on the optimization of large language model-based agents](#). *Preprint*, arXiv:2503.12434.
- Lishui Fan, Zhongxin Liu, Haoye Wang, Lingfeng Bao, Xin Xia, and Shanping Li. 2025. [Fait: Fault-aware fine-tuning for better code generation](#). *Preprint*, arXiv:2503.16913.
- Tom Gunter, Zirui Wang, Chong Wang, Ruoming Pang, Andy Narayanan, Aonan Zhang, Bowen Zhang, Chen Chen, Chung-Cheng Chiu, David Qiu, Deepak Gopinath, Dian Ang Yap, Dong Yin, Feng Nan, Floris Weers, Guoli Yin, Haoshuo Huang, Jianyu Wang, Jiarui Lu, and 136 others. 2024. [Apple intelligence foundation language models](#). *Preprint*, arXiv:2407.21075.
- Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. [Deepseek-coder: When the large language model meets programming – the rise of code intelligence](#). *Preprint*, arXiv:2401.14196.
- Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. [Measuring coding challenge competence with apps](#). *Preprint*, arXiv:2105.09938.
- Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. [Metagpt: Meta programming for a multi-agent collaborative framework](#). *Preprint*, arXiv:2308.00352.
- Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin de Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. 2019. [Parameter-efficient transfer learning for nlp](#). *Preprint*, arXiv:1902.00751.
- 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.
- Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. 2024. [Agentcoder: Multi-agent-based code generation with iterative testing and optimisation](#). *Preprint*, arXiv:2312.13010.
- Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Qian, and 5 others. 2024. [Qwen2.5-coder technical report](#). *Preprint*, arXiv:2409.12186.
- Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2024. [MapCoder: Multi-agent code generation for competitive problem solving](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4912–4944, Bangkok, Thailand. Association for Computational Linguistics.
- Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. 2025. [Codesim: Multi-agent code generation and problem solving through simulation-driven planning and debugging](#). *Preprint*, arXiv:2502.05664.
- Nan Jiang, Xiaopeng Li, Shiqi Wang, Qiang Zhou, Soneya Binta Hossain, Baishakhi Ray, Varun Kumar,

- Xiaofei Ma, and Anoop Deoras. 2024a. [Ledex: Training llms to better self-debug and explain code](#). In *Advances in Neural Information Processing Systems*, volume 37, pages 35517–35543. Curran Associates, Inc.
- Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2024b. [Self-planning code generation with large language models](#). *Preprint*, arXiv:2303.06689.
- Leela S. Karumbunathan. 2022. NVIDIA Jetson AGX Orin Series Technical Brief. Technical Report TB\_10749-001\_v1.2, NVIDIA Corporation. Version 1.2.
- Mohammad Abdullah Matin Khan, M Saiful Bari, Xuan Long Do, Weishi Wang, Md Rizwan Parvez, and Shafiq Joty. 2024. [XCodeEval: An execution-based large scale multilingual multitask benchmark for code understanding, generation, translation and retrieval](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6766–6805, Bangkok, Thailand. Association for Computational Linguistics.
- Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. 2023. [Efficient memory management for large language model serving with pagedattention](#). *Preprint*, arXiv:2309.06180.
- Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022. [Competition-level code generation with alphacode](#). *Science*, 378(6624):1092–1097.
- Xuechen Liang, Yangfan He, Meiling Tao, Yinghui Xia, Jianhui Wang, Tianyu Shi, Jun Wang, and JingSong Yang. 2025. [Cmat: A multi-agent collaboration tuning framework for enhancing small language models](#). *Preprint*, arXiv:2404.01663.
- OpenAI, :, Ahmed El-Kishky, Alexander Wei, Andre Saraiva, Borys Minaiev, Daniel Selsam, David Dohan, Francis Song, Hunter Lightman, Ignasi Clavera, Jakub Pachocki, Jerry Tworek, Lorenz Kuhn, Lukasz Kaiser, Mark Chen, Max Schwarzer, Mostafa Rohaninejad, Nat McAleese, and 7 others. 2025. [Competitive programming with large reasoning models](#). *Preprint*, arXiv:2502.06807.
- OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, and 262 others. 2024. [Gpt-4 technical report](#). *Preprint*, arXiv:2303.08774.
- Ruwei Pan, Hongyu Zhang, and Chao Liu. 2025. [Codecor: An llm-based self-reflective multi-agent framework for code generation](#). *Preprint*, arXiv:2501.07811.
- Shanghaoran Quan, Jiayi Yang, Bowen Yu, Bo Zheng, Dayiheng Liu, An Yang, Xuancheng Ren, Bofei Gao, Yibo Miao, Yunlong Feng, Zekun Wang, Jian Yang, Zeyu Cui, Yang Fan, Yichang Zhang, Binyuan Hui, and Junyang Lin. 2025. [Codeelo: Benchmarking competition-level code generation of llms with human-comparable elo ratings](#). *Preprint*, arXiv:2501.01257.
- Qwen, :, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiayi Yang, Jingren Zhou, and 25 others. 2025. [Qwen2.5 technical report](#). *Preprint*, arXiv:2412.15115.
- Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, and 7 others. 2024. [Code llama: Open foundation models for code](#). *Preprint*, arXiv:2308.12950.
- Weizhou Shen, Chenliang Li, Hongzhan Chen, Ming Yan, Xiaojun Quan, Hehong Chen, Ji Zhang, and Fei Huang. 2024. [Small LLMs are weak tool learners: A multi-LLM agent](#). In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 16658–16680, Miami, Florida, USA. Association for Computational Linguistics.
- Jiabin Tang, Tianyu Fan, and Chao Huang. 2025. [Autoagent: A fully-automated and zero-code framework for llm agents](#). *Preprint*, arXiv:2502.05957.
- Yun-Da Tsai, Mingjie Liu, and Haoxing Ren. 2024. [Code less, align more: Efficient llm fine-tuning for code generation with data pruning](#). *Preprint*, arXiv:2407.05040.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2023. [Chain-of-thought prompting elicits reasoning in large language models](#). *Preprint*, arXiv:2201.11903.
- Congying Xia, Chen Xing, Jiangshu Du, Xinyi Yang, Yihao Feng, Ran Xu, Wenpeng Yin, and Caiming Xiong. 2024. [Fofo: A benchmark to evaluate llms’ format-following capability](#). *Preprint*, arXiv:2402.18667.
- Wenda Xu, Rujun Han, Zifeng Wang, Long T. Le, Dhruv Madeka, Lei Li, William Yang Wang, Rishabh Agarwal, Chen-Yu Lee, and Tomas Pfister. 2025. [Speculative knowledge distillation: Bridging the teacher-student gap through interleaved sampling](#). *Preprint*, arXiv:2410.11325.

- Dayu Yang, Antoine Simoulin, Xin Qian, Xiaoyi Liu, Yuwei Cao, Zhaopu Teng, and Grey Yang. 2025. [Docagent: A multi-agent system for automated code documentation generation](#). *Preprint*, arXiv:2504.08725.
- Michihiro Yasunaga, Xinyun Chen, Yujia Li, Panupong Pasupat, Jure Leskovec, Percy Liang, Ed H. Chi, and Denny Zhou. 2024. [Large language models as analogical reasoners](#). *Preprint*, arXiv:2310.01714.
- Yongda Yu, Guoping Rong, Haifeng Shen, He Zhang, Dong Shao, Min Wang, Zhao Wei, Yong Xu, and Juhong Wang. 2024. [Fine-tuning large language models to improve accuracy and comprehensibility of automated code review](#). *ACM Trans. Softw. Eng. Methodol.*, 34(1).
- Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. 2022. Star: self-taught reasoner bootstrapping reasoning with reasoning. In *Proceedings of the 36th International Conference on Neural Information Processing Systems, NIPS '22*, Red Hook, NY, USA. Curran Associates Inc.
- Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. 2023. [Agenttuning: Enabling generalized agent abilities for llms](#). *Preprint*, arXiv:2310.12823.
- Chi Zhang, Zhao Yang, Jiaxuan Liu, Yucheng Han, Xin Chen, Zebiao Huang, Bin Fu, and Gang Yu. 2023. [Appagent: Multimodal agents as smartphone users](#). *Preprint*, arXiv:2312.13771.
- Wanjia Zhao, Mert Yuksekgonul, Shirley Wu, and James Zou. 2025. [Sirius: Self-improving multi-agent systems via bootstrapped reasoning](#). *Preprint*, arXiv:2502.04780.

## A Details of Supervisor

Fig. 7 and Fig. 8 illustrate the operation of the supervisor in our data curation pipeline. The supervisor receives the full execution trajectory of a problem, including the problem description, intermediate agent outputs (e.g., algorithm explanation, step-by-step plan, and code), and the result of test case execution.

As shown in Fig. 7, the input prompt instructs the supervisor to analyze the trajectory, determine which agent is responsible for the failure, and provide natural language feedback targeting that agent. The goal is to isolate the error at the correct stage (retrieval, planning, or coding), and produce actionable guidance that enables re-generation only the faulty component.

Fig. 8 shows a representative response from the supervisor. It correctly attributes the failure to the planning agent, explains why the plan is insufficient, and suggests how it should be modified. This output is then used to re-invoke the corresponding agent with additional guidance, creating high-quality training data without human annotation.

## B Backbone Model Comparison: General-Purpose vs. Coder

To investigate the impact of backbone model selection in multi-agent code generation, we conducted a series of controlled experiments comparing general-purpose and code-specialized models. Specifically, we tested Qwen2.5-7B-Instruct (general-purpose) and Qwen2.5-Coder-7B-Instruct (code-specialized) under various configurations of the MapCoder and MapCoder-Lite pipelines. The results show that general-purpose models are more robust across all agent roles and better suited for both zero-shot and fine-tuned multi-agent setups.

**Benchmark Performance (Zero-shot).** Table 9 shows the performance of MapCoder using each backbone model. The general-purpose variant consistently achieves higher accuracy and fewer format failures, highlighting its superiority in multi-agent coordination and instruction-following tasks.

**Agent-wise Ablation (Zero-shot).** As shown in Table 10, replacing any agent in the general-purpose pipeline with its coder counterpart leads to a consistent accuracy drop, most notably in the planning and coding agents. Even for the coding role, where the coder model is expected to excel, performance decreases—highlighting the importance of upstream integration and general reason-

Benchmark	Metric	7B	7BCoder
xCodeEval (106)	Accuracy (%)↑	13.21	8.49
	Format Fails↓	29	44
APPS (150)	Accuracy (%)↑	6.00	2.49
	Format Fails↓	30	66
CodeContests (165)	Accuracy (%)↑	6.06	5.45
	Format Fails↓	59	77

Table 9: Comparison of Qwen2.5-7B-Instruct and Qwen2.5-Coder-7B-Instruct across benchmarks.

Retrieval	Planning	Coding	Debugging	xCodeEval (%)
7B	7B	7B	7B	13.21
7BCoder	7B	7B	7B	12.26
7B	7BCoder	7B	7B	10.38
7B	7B	7BCoder	7B	10.38
7B	7B	7B	7BCoder	12.26

Table 10: Ablation of 7B vs. 7B Coder in retrieval, planning, coding, and debugging agents on the xCodeEval benchmark.

ing.

**Training Loss after Fine-tuning.** We fine-tuned both models using identical LoRA settings across all agents. As summarized in Table 11, the coder model converges to higher final loss values, suggesting a poorer fit to role-specialized training data. This pattern is consistent across all agents.

**Ablation after LoRA Fine-tuning.** Table 12 presents accuracy when substituting each LoRA-finetuned agent with its coder-based variant. Accuracy declines in all settings, and the fully coder-based pipeline achieves only 18.87%, compared to 28.30% for the general-purpose variant. These results indicate that even after fine-tuning, general-purpose models better support the multi-agent workflow.

## C Evaluation on Recent Reasoning Models (Qwen3)

We evaluate MapCoder-Lite on Qwen3-4B, a recently released small language model that significantly improves single-agent reasoning and outperforms Qwen2.5-7B under direct prompting. As shown in Table 13, Qwen3-4B achieves strong direct-prompting performance (22.64% without thinking and 26.42% with thinking). However, this improvement does not directly translate into reliable multi-agent behavior. When deployed in the MapCoder pipeline without fine-tuning, Qwen3-4B attains only 3.77% accuracy, exhibiting the same instability observed with earlier backbones.

Model	Retrieval	Planning	Coding	Debugging
Qwen2.5-7B-Instruct	0.34	0.25	0.08	0.33
Qwen2.5-Coder-7B-Instruct	0.36	0.35	0.13	0.34

Table 11: Final training loss after LoRA fine-tuning for each agent using general-purpose vs. coder-specific models.

Retrieval	Planning	Coding	Debugging	xCodeEval (%)
7BFT	7BFT	7BFT	7BFT	28.30
7BCoderFT	7BFT	7BFT	7BFT	21.70
7BFT	7BCoderFT	7BFT	7BFT	23.58
7BFT	7BFT	7BCoderFT	7BFT	24.53
7BFT	7BFT	7BFT	7BCoderFT	23.58
7BCoderFT	7BCoderFT	7BCoderFT	7BCoderFT	18.87

Table 12: xCodeEval accuracy when substituting each agent with a LoRA-finetuned coder-specific model (7BCoderFT), compared against the general-purpose baseline (7BFT).

Applying MapCoder-Lite mitigates this issue. After fine-tuning the retrieval, planning, and coding agents, Qwen3-4B achieves 30.19% accuracy, surpassing the fine-tuned Qwen2.5-7B despite using fewer parameters. These results indicate that recent advances in single-agent reasoning alone are insufficient for stable multi-agent workflows, and that targeted agent-wise specialization remains essential. Moreover, the larger gains observed with Qwen3-4B suggest that MapCoder-Lite scales positively with backbone capability, effectively activating role-specific behaviors that are not induced by prompting alone.

## D Generalization Across Model Families

While our main experiments used the Qwen series due to their strong coding performance on HumanEval and MBPP during development, our method remains architecture-agnostic. Since MapCoder-Lite applies LoRA through the PEFT library, it can be seamlessly used with any compatible model.

To test this generalizability, we evaluated both MapCoder and MapCoder-Lite using Llama3.1-8B-Instruct and CodeLlama-7B-Instruct-hf as backbones. As shown in Table 14, both models performed poorly without fine-tuning on competitive programming tasks. For example, Llama3.1-8B scored 0.00% with 94 format failures on xCodeEval. After applying MapCoder-Lite, Llama3.1-8B achieved 16.04% accuracy with zero format failures, demonstrating the adaptability of our pipeline. In contrast, CodeLlama-7B showed limited improvement on xCodeEval but responded well on

Method	Qwen2.5-7B-Instruct	Qwen3-4B
Direct (non-thinking)	18.87	22.64
Direct (thinking)	–	26.42
MapCoder (RPC)	11.32	3.77
MapCoder-Lite (RPC)	22.64	<b>30.19</b>

Table 13: xCodeEval performance of Qwen2.5-7B and Qwen3-4B under direct prompting and MapCoder pipelines. RPC denotes the retrieval–planning–coding agents.

Model	Benchmark	MapCoder	MapCoder-Lite
		Accuracy (%)	Accuracy (%)
Llama3.1-8B-Instruct	xCodeEval	0.00	16.04
CodeLlama-7B-Instruct	xCodeEval	0.94	2.83
CodeLlama-7B-Instruct	HumanEval	10.98	45.12

Table 14: Comparison of MapCoder and MapCoder-Lite on Llama3.1-8B and CodeLlama-7B models on xCodeEval and HumanEval benchmarks.

function-level benchmarks such as HumanEval.

Additionally, we compare MapCoder-Lite with various prompting strategies using the Llama3.1-8B backbone. As summarized in Table 15, MapCoder underperforms direct prompting baselines, whereas MapCoder-Lite substantially improves accuracy and surpasses all prompting methods. These results reinforce that multi-agent pipelines require targeted fine-tuning to be effective: without it, they underperform simpler approaches, while with it, they become competitive and scalable.

## E Improvements After Fine-Tuning

Fig. 9, 10, 11, and 12 illustrate representative improvements observed in the 7B model for the retrieval, planning, coding, and debugging agents, respectively.

In the retrieval example (Fig. 9), the pre-fine-tuned model incorrectly identifies the core algorithm as “Sliding Window” and includes unsupported tags like <description> within the <algorithm> block, failing to conform to the expected XML schema. After fine-tuning, the agent accurately identifies the algorithm (“Counting and Matching Pairs”) and outputs a well-structured XML response with all required tags properly closed.

In the planning example (Fig. 10), the original plan fails to capture key logical conditions (e.g., that the input must be both even and greater than 2). After fine-tuning, the agent successfully generates a plan that handles these conditions explicitly and correctly guides the coding agent.

<b>Method</b>	<b>Accuracy (%)</b>
Direct	6.60
CoT	4.72
Self-planning	4.72
Analogical	5.66
Multi-agent (MapCoder)	0.00
Multi-agent + FT (MapCoder-Lite)	16.04

Table 15: Accuracy comparison between MapCoder-Lite and other prompting baselines on xCodeEval using Llama3.1-8B-Instruct.

In the coding example (Fig. 11), the unfine-tuned model incorrectly processes multiple input values on a single line, violating the problem’s input specification. Fine-tuning enables the agent to read inputs line by line, aligning its behavior with the expected input format and improving functional correctness.

Finally, the debugging example (Fig. 12) demonstrates how the unfine-tuned model fails to fix a bug caused by improper input parsing. The fine-tuned debugging agent correctly diagnoses the issue and proposes a revised plan and code that successfully passes all test cases.

## F Trajectory Example

These are example trajectories used for fine-tuning each agent. Each consists of an input–output pair, and the examples are drawn from different problems.

### F.1 Retrieval Agent

#### Example Trajectory of Retrieval (Input)

Given a problem, algorithm tags, and solution code, provide relevant problems then identify the algorithm behind it and also explain the tutorial of the algorithm.

# Problem:

You are given several queries. Each query consists of three integers  $p$ ,  $q$  and  $b$ . You need to answer whether the result of  $p/q$  in notation with base  $b$  is a finite fraction.

A fraction in notation with base  $b$  is finite if it contains finite number of numerals after the decimal point. It is also possible that a fraction has zero numerals after the decimal point.

—Input—

The first line contains a single integer  $n$  ( $1 \leq n \leq 10^5$ ) — the number of queries.

Next  $n$  lines contain queries, one per line. Each line contains three integers  $p$ ,  $q$ , and  $b$  ( $0 \leq p \leq 10^{18}$ ,  $1 \leq q \leq 10^{18}$ ,  $2 \leq b \leq 10^{18}$ ). All numbers are given in notation with base 10.

—Output—

For each question, in a separate line, print Finite if the fraction is finite and Infinite otherwise.

—Examples—

Input

2

6 12 10

4 3 10

Output

Finite

Infinite

Input

4

1 1 2

9 36 2

4 12 3

3 5 4

Output

Finite

Finite

Finite

Infinite

—Note—

$$\frac{6}{12} = \frac{1}{2} = 0.5_{10}$$

$$\frac{4}{3} = 1.(3)_{10}$$

$$\frac{9}{36} = \frac{1}{4} = 0.01_2$$

$$\frac{4}{12} = \frac{1}{3} = 0.1_3$$

Sample Input Format:

2

6 12 10

4 3 10

Sample Output Format:

Finite

Infinite

---

Important: You must follow the input output format. Input should be taken from standard input and output should be given to standard output.

Note: If you are writing a function then after the function definition take input from using input() function, call the function with specified parameters and finally print the output of the function.

# Exemplars:

Recall three (03) relevant and distinct problems (different from problem mentioned above). For each problem,

1. describe it
2. generate Python3 code step by step to solve that problem
3. finally generate a planning to solve that problem

# Algorithm:

---

Important:

Your response must follow the following xml format-

<root>

<problem>

# Recall three (03) relevant and distinct problems (different from problem mentioned above). Write each problem in the following format.

<description>

# Describe the problem.

</description>

<code>

# Let's think step by step to solve this problem in Python3 programming language.

</code>

<planning>

# Planning to solve this problem.

```
</planning>
</problem>
```

# similarly add more problems here...

```
<algorithm>
# Identify the algorithm (Brute-force, Dynamic Programming, Divide-and-conquer, Greedy,
Backtracking, Recursive, Binary search, and so on) that needs to be used to solve the original
problem.
# Write a useful tutorial about the above mentioned algorithms. Provide a high level generic tutorial
for solving this types of problem. Do not generate code.
</algorithm>
</root>
```

### Example Trajectory of Retrieval (Output)

```
<root>
<problem>
<description>
Given a fraction  $p/q$ , determine if it can be represented as a finite decimal in base  $b$ . The fraction is
finite in base  $b$  if the denominator  $q$  (after simplifying the fraction) has no prime factors other than
those of the base  $b$ .
</description>
<code>
import math

def solve():
    n = int(input())
    for _ in range(n):
        p, q, b = map(int, input().split())
        # Simplify the fraction p/q
        g = math.gcd(p, q)
        q_simplified = q // g
        # Remove all common factors between q_simplified and b
        while True:
            g = math.gcd(q_simplified, b)
            if g == 1:
                break
            q_simplified //= g
        if q_simplified == 1:
            print("Finite")
        else:
            print("Infinite")

solve()
</code>
<planning>
1. Read the number of queries.
2. For each query, read  $p$ ,  $q$ , and  $b$ .
3. Simplify the fraction  $p/q$  by dividing both numerator and denominator by their greatest common
divisor (GCD).
```

4. Repeatedly divide the simplified denominator by the GCD of the denominator and the base  $b$  until no more common factors exist.
5. If the resulting denominator is 1, the fraction is finite in base  $b$ ; otherwise, it is infinite.
6. Print "Finite" or "Infinite" based on the result.

</planning>

</problem>

(Other example problems...)

<algorithm>

The algorithm used to solve the original problem is based on number theory, specifically the properties of fractions and their representations in different bases. The key insight is that a fraction  $p/q$  has a finite representation in base  $b$  if and only if the denominator  $q$  (after simplifying the fraction) has no prime factors other than those of the base  $b$ .

Tutorial:

1. Simplify the fraction  $p/q$  by dividing both numerator and denominator by their GCD.
2. Remove all common prime factors between the simplified denominator and the base  $b$  by repeatedly dividing the denominator by the GCD of the denominator and the base.
3. If the resulting denominator is 1, the fraction is finite in base  $b$ ; otherwise, it is infinite.

This approach efficiently checks the finiteness of the fraction by leveraging GCD operations to eliminate common factors, ensuring optimal performance even for large inputs.

</algorithm>

</root>

## F.2 Planning Agent

### Example Trajectory of Planning (Input)

Given a competitive programming problem generate a concrete planning to solve the problem.

# Problem:

Given a string, reverse each word in the string while keeping the order of the words the same. For example, "Hello World" becomes "olleH dlroW".

# Planning:

1. Split the input string into individual words using the 'split()' method.
2. Iterate over each word and reverse it using slicing ('[::-1]').
3. Join the reversed words back into a single string with spaces separating them.
4. Print the result.

## Relevant Algorithm to solve the next problem:

The algorithm used to solve the original problem (extracting the first character of each word in a string) is a **String Manipulation** technique, specifically involving splitting the string into words and processing each word individually.

### Tutorial: String Manipulation for Word Processing

1. **Splitting the String**: Use the 'split()' method to divide the input string into a list of words based on spaces.
2. **Processing Each Word**: Iterate over the list of words and extract the first character of each word.

3. **\*\*Combining Results\*\***: Join the extracted characters into a single string to form the final output.

This approach is efficient for problems where you need to process individual words in a string. It leverages basic string operations and does not require complex algorithms like dynamic programming or backtracking. The key steps are splitting, iterating, and combining, which are fundamental in string manipulation tasks.

**## Problem to be solved:**

In this exercise, a string is passed to a method and a new string has to be returned with the first character of each word in the string.

For example:

```
""  
"This Is A Test" ==> "TIAT"  
""
```

**Sample Input Format:**

sees eyes xray yoat

**Sample Output Format:**

sexy

\_\_\_\_\_

**Important:** You must follow the input output format. Input should be taken from standard input and output should be given to standard output.

**Note:** If you are writing a function then after the function definition take input from using 'input()' function, call the function with specified parameters and finally print the output of the function.

**## Sample Test cases:**

**Input:**

sees eyes xray yoat

**Expected output:**

sexy

**Input:**

brown eyes are nice

**Expected output:**

bean

**## Planning:**

\_\_\_\_\_

**Important:** You should give only the planning to solve the problem. Do not add extra explanation or words.

### Example Trajectory of Planning (Output)

1. Split the input string into individual words using the 'split()' method.
2. Iterate over each word and extract the first character.
3. Join the extracted characters into a single string.
4. Print the result.

### F.3 Coding Agent

#### Example Trajectory of Coding (Input)

Given a competitive programming problem generate Python3 code to solve the problem.

## Relevant Algorithm to solve the next problem:

The algorithm used in the original problem is **String Slicing**.

### Tutorial: String Slicing

String slicing is a technique to extract parts of a string by specifying start, end, and step indices. The general syntax is `string[start:end:step]`, where:

- `start` is the index where the slice begins (inclusive).
- `end` is the index where the slice ends (exclusive).
- `step` determines the stride (e.g., 1 for forward, -1 for reverse).

#### Key Points:

1. **Indices**: Python uses zero-based indexing. Negative indices count from the end (`-1` is the last character).
2. **Defaults**: If omitted, `start` defaults to 0, `end` to `len(string)`, and `step` to 1.
3. **Applications**:
  - Extracting substrings (`s[1:4]`).
  - Reversing strings (`s[::-1]`).
  - Removing characters (`s[1:-1]` removes first and last characters).

#### High-Level Approach:

1. Identify the substring you need by determining the start and end indices.
2. Use slicing to extract the substring.
3. Handle edge cases (e.g., empty strings, out-of-bound indices).

This technique is efficient ( $O(k)$  for a slice of length  $k$ ) and widely used in string manipulation tasks.

## Problem to be solved:

It's pretty straightforward. Your goal is to create a function that removes the first and last characters of a string. You're given one parameter, the original string. You don't have to worry with strings with less than two characters.

Sample Input Format:

eloquent

Sample Output Format:

loquen

\_\_\_\_\_

Important: You must follow the input output format. Input should be taken from standard input and output should be given to standard output.

Note: If you are writing a function then after the function definition take input from using `input()` function, call the function with specified parameters and finally print the output of the function.

## Planning:

1. Define a function `remove_first_last_char` that takes a string `s` as input.
2. Use string slicing to exclude the first and last characters (`s[1:-1]`).
3. Take input from the user, call the function, and print the result.

## Sample Test cases:

Input:

eloquent

Expected output:

loquen

Input:

country

Expected output:

ountr

## Let's think step by step.

---

Important:

## Your response must contain only the Python3 code to solve this problem. Do not add extra explanation or words.

#### Example Trajectory of Coding (Output)

```
“python
def remove_first_last_char(s):
    return s[1:-1]
input_str = input()
print(remove_first_last_char(input_str))
“
```

## F.4 Debugging Agent

#### Example Trajectory of Debugging (Input)

Given a competitive programming problem you have generated Python3 code to solve the problem. But the generated code can not pass sample test cases. Improve your code to solve the problem correctly.

## Relevant Algorithm to solve the next problem:

The algorithm used to solve the original problem of breaking up camel casing involves a straightforward approach to identifying capital letters within a string and inserting a space before them. This can be categorized under the Brute-force method since it directly iterates through the string and applies a simple condition to achieve the desired outcome.

**\*\*Tutorial on Brute-force Algorithms:\*\***

Brute-force algorithms involve trying every possible solution until the correct one is found. They are often the simplest and most direct way to solve a problem but may not always be the most efficient, especially for larger datasets. The key aspects of implementing a brute-force algorithm include:

- Identifying all possible solutions.
- Checking each solution against the problem constraints.
- Selecting the first solution that meets all criteria.

In the context of the camel case problem, the brute-force approach directly iterates through the string, checks if a character is uppercase, and if so, inserts a space before it. This method is effective

for the task at hand due to its simplicity and directness.

## Problem to be solved:

Complete the solution so that the function will break up camel casing, using a space between words.

### Example

...

```
solution("camelCasing") == "camel Casing"
```

...

Sample Input Format:

helloWorld

Sample Output Format:

hello World

\_\_\_\_\_

Important: You must follow the input output format. Input should be taken from standard input and output should be given to standard output.

Note: If you are writing a function then after the function definition take input from using `input()` function, call the function with specified parameters and finally print the output of the function.

## Planning:

1. Identify the indices where a transition from lowercase to uppercase letter occurs.
2. Insert a space before each identified index.
3. Join the modified string segments.
4. Return or print the final result.

## Code:

...

```
def solution(s):  
    return ''.join([' ' + c if c.isupper() else c for c in s])[1:]
```

```
s = input()  
print(solution(s))
```

...

## Test Report:

## Tested passed:

No test cases passed.

## Tests failed:

Input:

helloWorld

Expected Output:

hello World

Your Output:

ello World

Input:

camelCase

Expected Output:

camel Case  
Your Output:  
amel Case

## Modified Planning:  
## Let's think step by step to modify Python3 Code for solving this problem.

---

Important:

## Note: Strictly follow the input and output format. The input should be taken from Standard input and output should be given to standard output. If you are writing a function then after the function definition take input using `input()` function then call the function with specified parameters and finally print the output of the function. Do not add extra print statement otherwise it will failed the test cases.

## Your response must contain the modified planning and then the Python3 code inside `"""` block to solve this problem.

### Example Trajectory of Debugging (Output)

## Modified Planning:

1. **Identify Uppercase Letters**: Iterate through each character in the input string to check if it is uppercase.
2. **Insert Space Before Uppercase Letters**: For each uppercase character found, insert a space before it.
3. **Construct Result String**: Combine all characters, including the inserted spaces, to form the final result string.
4. **Handle Edge Cases**: Ensure that the first character of the string is not modified if it is uppercase (no space should be inserted before the first character).

The issue with the original code was that it was slicing the string from the second character (`[1:]`), which caused the first character to be omitted. This needs to be fixed by ensuring the first character is included in the result.

## Modified Python3 Code:

```
"""python
def solution(s):
    result = []
    for c in s:
        if c.isupper():
            result.append(' ' + c)
        else:
            result.append(c)
    return ''.join(result)

s = input()
print(solution(s))
"""
```

You are an expert verifier in a multi-agent code generation system. The final code failed to pass the sample test cases. Your task is to carefully review the retrieved algorithm, plan, and code. Identify which part(s) were incorrect, and provide specific suggestions for improvement. Only include agents responsible for the mistake.

```

# Problem Description:
{verification_problem_desc}

# Retrieved Algorithm:
{verification_algorithm}

# Original Plan:
{planning}

# Generated Code:
\`
\`
{code}
\`
\`

# Test Log:
{test_log}

-----
Important:
- If a mistake in a later stage (e.g., Planning or Coding) was actually caused by a flawed output from an earlier stage (e.g., Retrieval or Planning), you MUST attribute the responsibility to the earlier one.
- For example: If the code fails because the plan was incorrect → Planning is responsible, not Coding.

- Your output MUST follow this strict XML format:
<root>
<responsible_agents>
<agent>Planning</agent> <!-- Or Retrieval / Coding / Multiple entries allowed -->
</responsible_agents>
<reasons>
<reason>Explain why this agent was responsible.</reason>
</reasons>
<suggestions>
<!-- Only include suggestions for the agents listed in <responsible_agents> -->
<suggestion target="\Retrieval\ ">Suggest improvement for Retrieval Agent.</suggestion>
<suggestion target="\Planning\ ">Suggest improvement for Planning Agent.</suggestion>
<suggestion target="\Coding\ ">Suggest improvement for Coding Agent.</suggestion>
</suggestions>
</root>

```

Figure 7: Prompt for the Supervisor.

```

<root>
  <responsible_agents>
    <agent><Planning></agent>
  </responsible_agents>
  <reasons>
    <reason><The planning agent incorrectly assumed that each team consists of 3 members (2 programmers and 1 mathematician or vice versa) and divided the minimum of a and b by 3. However, the problem requires each team to have exactly 4 members with at least one programmer and one mathematician. This flawed logic led to incorrect calculations for all test cases.></reason>
  </reasons>
  <suggestions>
    <suggestion target="Planning"><The planning agent should consider that each team must have exactly 4 members with at least one programmer and one mathematician. The correct approach is to calculate the maximum number of teams as the minimum of (a + b) // 4, a, and b. This ensures that the total number of students is sufficient for the teams, and there are enough programmers and mathematicians to meet the constraints.></suggestion>
  </suggestions>
</root>

```

Figure 8: Example response from the supervisor: identifies the flawed part of the plan and generates feedback.

Problem Description:

Ivan has a robot which is situated on an infinite grid. Initially the robot is standing in the starting cell (0, 0). The robot can process commands. There are four types of commands it can perform: U – move from the cell (x, y) to (x, y + 1); D – move from (x, y) to (x, y - 1); L – move from (x, y) to (x - 1, y); R – move from (x, y) to (x + 1, y). Ivan entered a sequence of n commands, and the robot processed it. After this sequence the robot ended up in the starting cell (0, 0), but Ivan doubts that the sequence is such that after performing it correctly the robot ends up in the same cell. He thinks that some commands were ignored by robot. To acknowledge whether the robot is severely bugged, he needs to calculate the maximum possible number of commands that were performed correctly. Help Ivan to do the calculations!



Retrieval  
(Qwen7B)

```
<root>
<problem>
...
</problem>
<algorithm>
The original problem can be solved using a
combination of **Sliding Window** and **Hash Map**
techniques.

The Sliding Window algorithm is a technique ...
</algorithm>
```

Wrong algorithm explanation,  
unclosed <root> tag



Retrieval  
(Qwen7B-FT)

```
<root>
<problem>
...
</problem>
<algorithm>
The algorithm used to solve the original problem involves
checking the balance between the occurrences of certain
pairs of commands ('U' and 'D', 'L' and 'R'). This
requires counting the number of times each type of command
appears and ensuring that they cancel out perfectly to
allow the robot to return to its starting position.

**Tutorial: Counting and Matching Pairs**
...
</algorithm>
<root>
```

Correct algorithm tutorial  
and format

Figure 9: Improvement in algorithm tutorial and XML formatting by the retrieval agent after fine-tuning. The pre-trained model fails to identify the correct algorithm and generates an ill-formed XML response with an unclosed <root> tag. After fine-tuning, the retrieval agent provides a valid explanation based on matching command pairs and generates a well-structured XML block.

Problem Description:

One hot summer day Pete and his friend Billy decided to buy a watermelon. They chose the biggest and the ripest one, in their opinion. After that the watermelon was weighed, and the scales showed w kilos. They rushed home, dying of thirst, and decided to divide the berry, however they faced a hard problem. Pete and Billy are great fans of even numbers, that's why they want to divide the watermelon in such a way that each of the two parts weighs even number of kilos, at the same time it is not obligatory that the parts are equal. The boys are extremely tired and want to start their meal as soon as possible, that's why you should help them and find out, if they can divide the watermelon in the way they want. For sure, each of them should get a part of positive weight.



Planning  
(Qwen7B)

1. Check if the weight 'w' is even.
2. If 'w' is even, return "Yes".
3. Otherwise, return "No".

w must be even and  
greater than 2



Planning  
(Qwen7B-FT)

1. Read the input value 'w'.
2. Check if 'w' is greater than 2 and divisible by 2.
3. If true, print "Yes".
4. Otherwise, print "No".

Fine-tuned planning agent generates  
conditions correctly

Figure 10: Improvement in conditional logic by the planning agent after fine-tuning. The initial code fails to account for the edge case (w > 2) and only checks for evenness. After fine-tuning, the planning agent correctly generates the full condition required by the problem.

**Problem Description:**  
 Nikolay has a lemons, b apples and c pears. He decided to cook a compote. According to the recipe the fruits should be in the ratio 1: 2: 4. It means that for each lemon in the compote should be exactly 2 apples and exactly 4 pears. You can't crumble up, break up or cut these fruits into pieces. These fruits – lemons, apples and pears – should be put in the compote as whole fruits. Your task is to determine the maximum total number of lemons, apples and pears from which Nikolay can cook the compote. It is possible that Nikolay can't use any fruits, in this case print 0.

**Input Specification:**  
The first line contains the positive integer a ( $1 \leq a \leq 1000$ ) – the number of lemons Nikolay has. The second line contains the positive integer b ( $1 \leq b \leq 1000$ ) – the number of apples Nikolay has. The third line contains the positive integer c ( $1 \leq c \leq 1000$ ) – the number of pears Nikolay has.

**Output Specification:**  
 Print the maximum total number of lemons, apples and pears from which Nikolay can cook the compote.

↓  
 Coding  
 (Qwen7B)

```

python
def max_fruits():
  a, b, c = map(int, input().split())
  min_val = min(a, b // 2, c // 4)
  total_fruits = min_val * 7
  print(total_fruits)

max_fruits()
python
  
```

Input values  
 should be read  
 separately

↓  
 Coding  
 (Qwen7B-FT)

```

python
def max_fruits(a, b, c):
  # Calculate the minimum number of sets based on the limiting factor
  min_sets = min(a, b // 2, c // 4)

  # Calculate the total number of fruits used
  total_fruits = min_sets * (1 + 2 + 4)

  return total_fruits

# Reading input
a = int(input())
b = int(input())
c = int(input())

# Calling the function and printing the result
print(max_fruits(a, b, c))
python
  
```

Fine-tuned coding agent reads  
 input values in the right way

Figure 11: Example of a coding error and its resolution after fine-tuning. The unfine-tuned agent incorrectly reads all input values from a single line, which violates the problem specification. After fine-tuning, the coding agent correctly processes line-separated inputs and computes the result accordingly.

```
# Problem:
Problem Description:
Nikolay has a lemons, b apples and c pears. He decided to cook a compote. According to the recipe the fruits should be in the ratio 1: 2: 4. It means that for each lemon in the compote should be exactly 2 apples and exactly 4 pears. You can't crumble up, break up or cut these fruits into pieces. These fruits – lemons, apples and pears – should be put in the compote as whole fruits. Your task is to determine the maximum total number of lemons, apples and pears from which Nikolay can cook the compote. It is possible that Nikolay can't use any fruits, in this case print 0.
Input Specification:
The first line contains the positive integer a (1 ≤ a ≤ 1000) – the number of lemons Nikolay has. The second line contains the positive integer b (1 ≤ b ≤ 1000) – the number of apples Nikolay has. The third line contains the positive integer c (1 ≤ c ≤ 1000) – the number of pears Nikolay has.
Output Specification:
Print the maximum total number of lemons, apples and pears from which Nikolay can cook the compote.
```



Retrieval,  
Planning, Coding

```
python
def max_fruites():
    a, b, c = map(int, input().split())
    min_val = min(a, b // 2, c // 4)
    total_fruites = 7 * min_val
    print(total_fruites)

max_fruites()
```

Input values  
should be read  
separately



Debugging  
(Qwen7B-FT)



Debugging  
(Qwen7B)

```
## Modified Planning:
1. Read Input: Take three integers as input representing the number of lemons, apples, and pears.
2. Calculate Maximum Sets: Determine the maximum number of complete sets of 1 lemon, 2 apples, and 4 pears that can be formed. This is determined by the minimum value among `a // 1`, `b // 2`, and `c // 4`.
3. Compute Total Fruits: Calculate the total number of fruits used based on the maximum number of complete sets.
4. Output Result: Print the total number of fruits used.

## Code:
python
def max_fruites():
    # Read input values for lemons, apples, and pears
    a, b, c = map(int, input().split())

    # Calculate the maximum number of complete sets
    min_val = min(a, b // 2, c // 4)

    # Compute the total number of fruits used
    total_fruites = 7 * min_val

    # Output the total number of fruits used
    print(total_fruites)

# Call the function and take input
max_fruites()
```

Fail to identify the  
problem

## Modified Planning:

After analyzing the error message, it becomes clear that the issue arises because the input is not being read correctly. Each line of input needs to be processed separately, and the input should be read line by line rather than trying to parse everything at once. We need to read each line individually and process the values accordingly.

Here's the revised plan:

1. Read the first line containing the number of lemons.
2. Read the second line containing the number of apples.
3. Read the third line containing the number of pears.
4. For each set of inputs, calculate the minimum value among the quotients obtained by dividing the respective fruit counts by their required quantities in the ratio (i.e., lemons by 1, apples by 2, and pears by 4).
5. Compute the total number of fruits used based on the minimum value calculated.
6. Print the result for each set of inputs.

This approach ensures that the input is read correctly and processed line by line, avoiding the issues seen before.

## Python3 Code:

```
python
def max_fruites():
    # Reading input values for lemons, apples, and pears
    a = int(input().strip()) # Number of lemons
    b = int(input().strip()) # Number of apples
    c = int(input().strip()) # Number of pears

    # Calculating the minimum value among the quotients
    min_val = min(a, b // 2, c // 4)

    # Computing the total number of fruits used
    total_fruites = 7 * min_val

    # Printing the result
    print(total_fruites)

# Calling the function to execute
max_fruites()
```

Debugging  
success

Figure 12: Example of a debugging failure and recovery in the MapCoder pipeline. The initial plan and code fail to address the problem due to incorrect input parsing. After the debugging agent identifies the issue, the revised plan introduces line-by-line input reading, which successfully resolves the error.