PairCoder: Pair Programming-Inspired Two-Agent Collaboration for Code Generation

Junhao Chen[1] **Xiang Li**[2] **Yibin Xu**[3] **Yuehan Cui**[4] **Fangsheng Weng**[5]
Hao Zhao[4,6] **Fei Ma**[7 †] **Qi Tian**[7]

[1]Shenzhen International Graduate School, Tsinghua University [2]Peking University
[3]Tongji University [4]AIR, Tsinghua University [5]Independent Researcher
[6]BAAI [7]Guangdong Laboratory of Artificial Intelligence and Digital Economy (SZ)

Abstract

Large Language Models (LLMs) achieve strong results on code generation, but single model inference remains brittle on tasks that require iterative refinement. Existing multi agent frameworks improve reliability, yet they often incur substantial token and latency overhead. We introduce PairCoder, a framework that brings pair programming to autonomous LLM collaboration. PairCoder assigns one model to code generation and the other to review, and switches roles when repeated errors suggest that the current interaction has stalled. Across 13 LLMs on HumanEval, PairCoder consistently improves over single model inference. On eight representative backbones, it reaches 91.0% pass@1 and improves over single model inference by up to 20.3% while reducing token usage by 40% to 70% relative to multi agent baselines. Many heterogeneous pairings also outperform both constituent models, suggesting that the framework generalizes across model families. These results position PairCoder as an effective and deployment conscious alternative to heavier multi agent systems. Code is available at https://github.com/yisuanwang/PairCoder.

1 Introduction

Automatic code generation, which maps natural language specifications to executable programs, is a central problem in software engineering with broad practical impact (Chen et al., 2021; Liu et al., 2024b; Jiang et al., 2024). Large Language Models (LLMs) such as GPT-4 (OpenAI, 2023), Qwen (Bai et al., 2023; Hui et al., 2024; Yang et al., 2025), DeepSeek (Liu et al., 2024a; Guo et al., 2025a), and StarCoder (Li et al., 2023a; Lozhkov et al., 2024) have pushed benchmark performance to impressive levels and made code generation one of the most active application areas of foundation models. Beyond code, these models have shown broad applicability: from video understanding (Ye et al., 2024) and style guided image synthesis (Chen et al., 2023b) to interactive video generation (Chen et al., 2026a,c), and VLM driven 3D reconstruction (Sun et al., 2025; Chen et al., 2026d). Efficient inference for such models also remains an active research direction (Chen et al., 2023c). Yet single model inference remains fragile, even in code generation, where planning, debugging, and verification can be evaluated directly. This makes the domain a useful testbed for asking whether stronger AI systems require more agents or better interaction structure (Li et al., 2023b; Zhang et al., 2025; Tian et al., 2025).

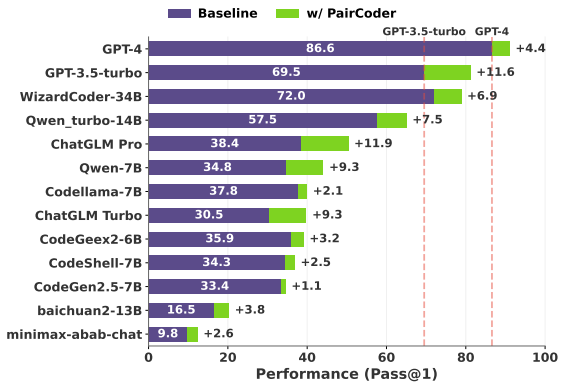

Figure 1: Performance comparison of 13 language models on code generation tasks (HumanEval Pass@1) showing consistent improvements with PairCoder (**green**) over baseline (**purple**).

Multi agent frameworks address this limitation by introducing multiple LLMs that collaborate, debate, or review one another's outputs. For example, MetaGPT (Hong et al., 2023) and ChatDev (Qian et al., 2023) simulate full software teams with structured role hierarchies, while AgentCoder (Huang et al., 2023), Self-Collaboration (Dong et al., 2024), AgentVerse (Chen et al., 2023d), and MapCoder (Islam et al., 2024) use lighter agent configurations to improve generation quality. These systems often improve correctness, but they do so at substantial cost. Simulating a full software team requires many

† Corresponding Author.

Findings of the Association for Computational Linguistics: ACL 2026, pages 3043–3058

agents and interaction rounds, which can drive token consumption to an order of magnitude above single model inference. Such overhead limits practical deployment, especially in production settings.

We therefore ask a simpler question: what is the smallest collaborative structure that can still deliver meaningful gains? We argue that the answer is already familiar from human software engineering, namely **pair programming** (Umapathy and Ritzhaupt, 2017; Lui and Chan, 2006). In pair programming, one developer writes code while the other continuously reviews it, spots errors, and steers the solution. Decades of empirical work show that this two person structure can reduce defects and improve code quality without the overhead of a full team (Hannay et al., 2009; Williams et al., 2000; Di Bella et al., 2012). If this structure transfers to LLMs, role allocation may matter more than agent count.

Motivated by this observation, we propose **PairCoder**, a two agent LLM framework inspired by pair programming. One agent acts as the Driver and proposes code, while the other acts as the Navigator and reviews it, diagnoses problems, and returns targeted feedback. The Driver then revises the candidate solution accordingly. When repeated errors indicate that progress has stalled, PairCoder switches the roles of the two agents. Unlike self improvement methods that rely on a single model to both generate and critique code (Shinn et al., 2024; Chen et al., 2024), PairCoder separates these responsibilities and introduces an external perspective at each iteration.

Across 13 LLMs on HumanEval (Chen et al., 2021), PairCoder consistently improves over single model inference. On eight representative backbones, it reaches 91.0% pass@1, improving over single model inference by up to 20.3% while reducing token consumption by 40% to 70% relative to existing multi agent baselines. Many heterogeneous pairs of weaker models also outperform both constituent models, suggesting that the collaboration pattern generalizes across model families.

Our main contributions are as follows: (1) we introduce PairCoder, a minimal two agent framework for LLM code generation grounded in pair programming; (2) we show that this collaboration pattern matches or exceeds stronger multi agent baselines while using substantially fewer tokens; and (3) we analyze role switching and find that error triggered exchange helps sustain progress during collaborative inference.

2 Related Work

2.1 Large Language Models

LLMs such as GPT-4 (OpenAI, 2023), LLaMA (Touvron et al., 2023), Qwen (Bai et al., 2023; Hui et al., 2024; Yang et al., 2025), and DeepSeek (Liu et al., 2024a; Guo et al., 2025a) have achieved strong performance across a broad range of NLP tasks (Bowman et al., 2015; Wei et al., 2022), and code generation has become one of their most impactful application domains (Vaithilingam et al., 2022; Li et al., 2023b). These models have also been applied to multimodal settings such as temporally consistent dense prediction (Miao et al., 2026) and temporal video grounding (Chenzhaoyu et al., 2026). Robustness issues arise in these settings as well, for example under VLM token pruning (Huang et al., 2026) and GUI agent evaluation (Chen et al., 2026e). Despite impressive benchmark results, single model inference remains vulnerable to logical errors and hallucinations (Zhang et al., 2025; Tian et al., 2025; Ji et al., 2023), which motivates more reliable collaborative inference strategies.

2.2 Code Generation with LLMs

Code generation has progressed from early syntax aware neural models (Ling et al., 2016; Yin and Neubig, 2017; Sun et al., 2020) to large pretrained systems such as Codex (Chen et al., 2021), CodeGen (Nijkamp et al., 2023), CodeGeeX (Zheng et al., 2023), StarCoder (Li et al., 2023a; Lozhkov et al., 2024), and CodeLlama (Roziere et al., 2023). These models benefit from large scale pretraining on code corpora and perform strongly on standard benchmarks. Related work has also extended token based generation to structured visual outputs such as vector animation (Chen et al., 2026b) and garment patterns (Weng et al., 2026). Although these tasks differ from executable program synthesis, they similarly require long range structural consistency under autoregressive decoding. However, evaluation on more realistic settings such as SWE-bench (Jimenez et al., 2024) shows that even top systems struggle on complex repository level tasks. Self improvement methods such as Reflexion (Shinn et al., 2024) and self debugging (Chen et al., 2024) allow a single model to revise its output using execution feedback, but they remain limited by the blind spots of a single perspective. PairCoder addresses this limitation by separating generation and review across two agents.

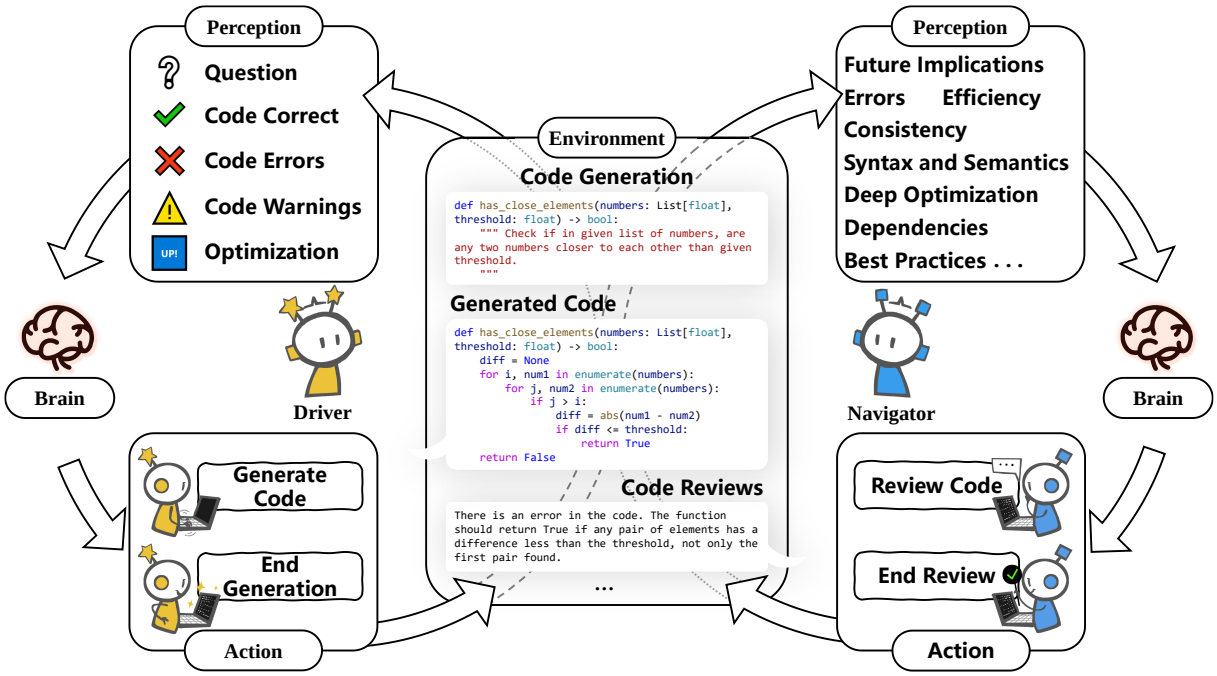

Figure 2: The PairCoder framework showing iterative collaboration between Driver (code generation) and Navigator (review and feedback) agents, with an automatic role switching mechanism.

2.3 Multi Agent Systems

Several multi agent systems improve code quality by dividing labor across specialized roles. MetaGPT (Hong et al., 2023) encodes a software team hierarchy, ChatDev (Qian et al., 2023) simulates a development company with waterfall style handoffs, AgentCoder (Huang et al., 2023) separates generation from test execution, Self-Collaboration (Dong et al., 2024) has one LLM play multiple roles sequentially, MapCoder (Islam et al., 2024) guides generation through planning and retrieval agents, and AgentVerse (Chen et al., 2023d) provides a general collaboration scaffold. Agent collaboration has also been explored beyond code generation, for example in multimodal 3D creation from interleaved inputs (Chen et al., 2025a). These approaches often outperform single model inference, but their reliance on large agent pools and many interaction rounds leads to token costs that can exceed a single call by an order of magnitude. PairCoder instead focuses on the smallest collaborative unit that still yields consistent gains, and in our experiments reduces this overhead by 40% to 70% in practice.

2.4 Pair Programming

Pair programming, in which one developer writes code while the other reviews it in real time, is one of the most studied practices in agile software engineering (Williams et al., 2000; Cockburn and Williams, 2000; Lui and Chan, 2006; Umapathy and Ritzhaupt, 2017). Controlled experiments and industrial case studies consistently report lower defect rates and faster knowledge transfer than solo programming (Hannay et al., 2009; Vanhanen and Lassenius, 2005; Di Bella et al., 2012). Recent work has explored collaboration between humans and AI models, where an LLM serves as a coding assistant (Chen et al., 2023a; Wu et al., 2023; Sarkar et al., 2022), but the model remains a passive tool rather than an autonomous reviewer. PairCoder is, to our knowledge, the first method to instantiate the Driver and Navigator pattern as a fully autonomous two LLM system in which both agents contribute to the final solution.

3 Methods

3.1 Overview

The PairCoder framework, illustrated in Fig. 2, adapts pair programming to automated code generation with Large Language Models (LLMs). Two agents alternate between proposal and review. At each iteration, the Driver produces a candidate solution and the Navigator critiques it, after which the Driver revises the code. This interaction continues until the solution is accepted or the iteration budget is exhausted.

Formally, given a programming task $q \in \mathcal{Q}$,

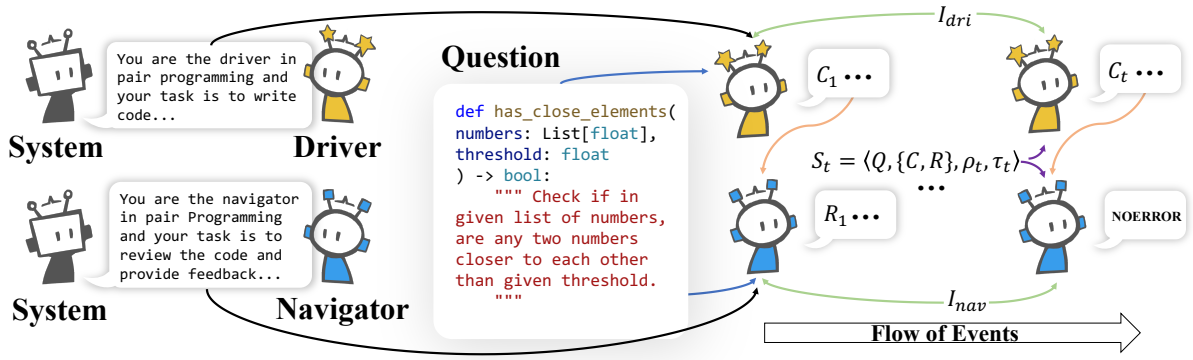

Figure 3: PairCoder workflow: code generation based on instructions, query, and historical context.

PairCoder seeks to generate code $c^* \in \mathcal{C}$ that maximizes both correctness and quality:

$$c^* = \arg\max_{c \in \mathcal{C}} P(c \mid q) \cdot \text{Quality}(c, q) \quad (1)$$

3.2 Role Assignment

At each time step, a role function $\rho : \mathcal{A} \times T \to \mathcal{R}$ assigns Driver and Navigator to the two agents, where $\mathcal{A} = \{a_1, a_2\}$ denotes the agents, T the time steps, and $\mathcal{R} = \{Driver, Navigator\}$ the available roles.

The Driver is responsible for code construction and revision, whereas the Navigator checks correctness, identifies deficiencies, and returns actionable feedback. When repeated revision signals or the iteration budget indicate that progress has stalled, the framework swaps the roles of the two agents so that the alternate model can continue the search.

Driver Agent. The Driver generates code conditioned on the task specification and interaction history. As illustrated in Fig. 3, its generation process follows:

$$C_t = F_{driver}(I_{dri}, Q, M_{t-1}) \quad (2)$$

where I_{dri} denotes the driver instructions, Q is the task description, C_t is the candidate code at iteration t, R_i is the review outcome at iteration i, and M_{t-1} is the accumulated interaction history:

$$M_t = \{(C_i, R_i) \mid 1 \le i \le t\} \quad (3)$$

The Driver aims to preserve semantic alignment with the task while incorporating the Navigator's latest feedback:

$$\mathcal{S}_{dri}(C_t) = \lambda_1 \cdot \mathcal{L}_{sem}(C_t, Q) + \lambda_2 \cdot \mathcal{L}_{adapt}(C_t, R_{t-1}) \quad (4)$$

where $\mathcal{L}_{sem}(\cdot)$ measures semantic alignment and $\mathcal{L}_{adapt}(\cdot)$ quantifies feedback integration.

Navigator Agent The Navigator evaluates the current candidate and returns structured feedback:

$$R_t = F_{navigator}(I_{nav}, Q, C_t, M_{t-1}) \quad (5)$$

where I_{nav} specifies navigator specific review guidelines.

The Navigator checks syntax, functional consistency, and general code quality:

$$R_t = \begin{cases} \text{ACCEPT} & \text{if } \bigwedge_{i=1}^{n} \psi_i(C_t) = \text{true} \\ \text{REVISE}(\Delta_t) & \text{otherwise} \end{cases} \quad (6)$$

where ψ_i represents verification predicates and Δ_t contains concrete revision suggestions.

The Navigator signals completion by issuing `[NOERROR]` when the stopping criteria are met. The interaction history is then updated as:

$$M_t = M_{t-1} \cup \{(C_t, R_t)\} \quad (7)$$

3.3 System Components

Shared Environment. The framework maintains a shared state $S_t = \langle Q, M_t, \rho_t, \tau_t \rangle$ with four components. The query Q is the original task specification. The memory M_t stores the interaction history with bounded size $|M_t| \le W$. The role assignment ρ_t records the current mapping between agents and roles, and the iteration counter τ_t tracks collaboration progress.

Perception Module. The perception module transforms external inputs into internal representations through role specific processing. The Driver uses $\Phi_d : (Q, M_{t-1}) \to$ CodeSpace to process the task and memory, while the Navigator uses $\Phi_n : (C_t, Q) \to$ FeedbackSpace to analyze the current code candidate. Both functions incorporate domain knowledge from software engineering

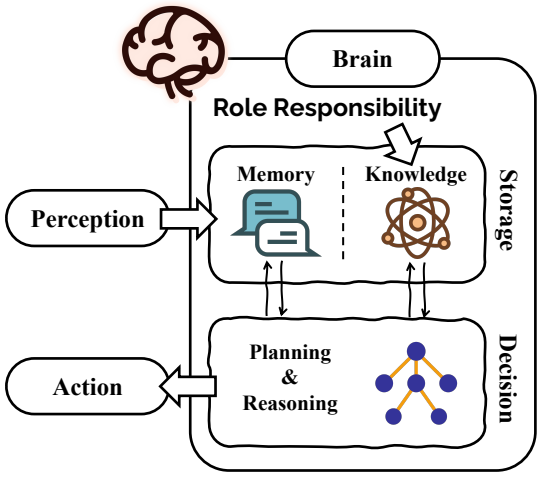

Figure 4: Agent brain architecture: LLM based decision making with role specific configuration.

best practices (McConnell, 2004; Fowler, 2018; Williams and Kessler, 2003; Martin, 2009).

Agent Brain. As depicted in Fig. 4, each agent uses an LLM conditioned on a role specific prompt and the current shared state:

$$\text{Brain}_{\text{role}} = \text{LLM}(P_{\text{role}}, S_t) \tag{8}$$

where P_{role} encodes the responsibilities of the current role and S_t provides the task context, dialogue history, and current role assignment.

Action Space. Each agent operates within a discrete action space $\mathcal{A}_{\text{role}}$:

$$\mathcal{A}_{driver} = \{\texttt{GenerateCode()}, \texttt{RefineCode()}, \texttt{EndGeneration()}\} \tag{9}$$

$$\mathcal{A}_{navigator} = \{\texttt{ReviewCode()}, \texttt{AcceptCode()}\} \tag{10}$$

3.4 Collaboration Protocol

The collaboration follows an iterative refinement process formalized in Algorithm 1. The feedback loop, illustrated in Fig. 5, alternates between Navigator review and Driver revision, yielding progressive improvement over successive iterations.

Termination Protocol. The system terminates when condition $\Omega(S_t)$ is satisfied:

$$\Omega(S_t) = \begin{cases} \text{true} & \text{if } R_t = \text{ACCEPT} \\ \text{true} & \text{if } \tau \geq T \\ \text{false} & \text{otherwise} \end{cases} \tag{11}$$

Algorithm 1 PairCoder Collaboration Protocol

Input: Task Q, parameters $\Theta = \{T, W, k, \eta\}$
Output: Optimized code C^*
Initialize $S_0 \leftarrow \langle Q, \emptyset, \rho_0, 0 \rangle$
while $\tau < T$ and not terminated **do**
  Driver phase with self mirror
  $P_{dri} \leftarrow \text{SelfMirror}(\text{Driver}, I_{dri})$
  $C_\tau \leftarrow F_{driver}(P_{dri}, Q, M_{\tau-1})$
  Navigator phase with self mirror
  $P_{nav} \leftarrow \text{SelfMirror}(\text{Navigator}, I_{nav})$
  $R_\tau \leftarrow F_{navigator}(P_{nav}, Q, C_\tau, M_{\tau-1})$
  if $R_\tau = \text{ACCEPT}$ **then**
    return C_τ
  end if
  Update shared state
  $M_\tau \leftarrow \text{Update}(M_{\tau-1}, C_\tau, R_\tau)$
  $\rho_{\tau+1} \leftarrow \text{CheckSwitch}(\rho_\tau, R_\tau, \tau)$
  $\tau \leftarrow \tau + 1$
end while
return $\arg\max_{C_i \in M_\tau} \text{Quality}(C_i)$

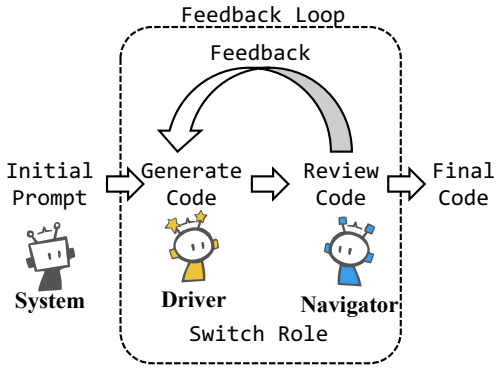

Figure 5: Collaboration mechanism with iterative feedback loop between Driver and Navigator agents.

Self Mirror Mechanism. To preserve role consistency, each agent receives an identity aware prompt before generation or review:

$$\text{SelfMirror}(\rho_t, Q, M_t) = \text{concat}(\text{RolePrompt}(\rho_t), Q, M_t) \tag{12}$$

where $\text{RolePrompt}(\rho_t)$ explicitly states whether the agent should act as Driver or Navigator. This mechanism keeps the Driver focused on code production and revision, and keeps the Navigator focused on diagnosis, verification, and targeted feedback throughout the interaction.

Role Switching Strategies. Our framework supports heterogeneous agent configurations in which different LLMs can serve as Driver and Naviga-

tor, allowing us to leverage complementary model strengths. To optimize collaboration dynamics in this setting, we implement two switching policies $\pi : S_t \to \{0, 1\}$:

Fixed interval switching.

$$\pi_{\text{fixed}}(S_t) = (\tau \bmod k = 0) \tag{13}$$

ensuring balanced participation every k iterations.

Error triggered switching.

$$\pi_{\text{error}}(S_t) = \left(\sum_{i=\tau-w}^{\tau} \mathbb{I}[R_i = \text{REVISE}] \geq \eta \right) \tag{14}$$

activating when error count exceeds threshold η within window w.

4 Results

4.1 Experimental Setup

Systematic LLM evaluation has been studied across many domains, from Chinese language understanding (Zhang et al., 2023) and strategic reasoning (Chen et al., 2025b) to web interface comprehension (Guo et al., 2025b). Following this tradition, we evaluate PairCoder on HumanEval (Chen et al., 2021), the standard benchmark for code generation. Our model pool includes 13 LLMs spanning proprietary systems (GPT-4 (OpenAI, 2023) and GPT-3.5 (OpenAI, 2023)), dialogue oriented models (the ChatGLM series (GLM et al., 2024) and minimax (MiniMax, 2023)), code specialized models (Codellama (Roziere et al., 2023), CodeGeex2 (Zheng et al., 2023), WizardCoder (Luo et al., 2023), CodeGen2.5 (Nijkamp et al., 2023), and StarCoder (Li et al., 2023a)), and general purpose models (baichuan2 (Baichuan, 2023) and the Qwen series (Bai et al., 2023)). We compare against seven prior multi agent methods (Huang et al., 2023; Dong et al., 2024; Hong et al., 2023; Chen et al., 2023d; Qian et al., 2023; Shinn et al., 2024; Nguyen et al., 2024). All experiments are conducted on 8 × A100 GPUs with dedicated allocation per LLM. Generated code is validated through regex based extraction and executability checks.

4.2 Role Switching Analysis

We evaluate two role switching strategies: ***Fixed Interval***, which swaps roles at predetermined rounds, and ***Error Triggered***, which swaps roles after consecutive revision signals from the Navigator. As shown in Fig. 6, the error triggered policy yields the best accuracy when the switch occurs after two consecutive error detections. Larger thresholds, such as ten rounds, bring performance close to single model behavior because an unproductive Driver remains in place for too long before the alternate agent can take over. Optimal switching thresholds vary across LLM pairs, but the two error setting is the most reliable overall.

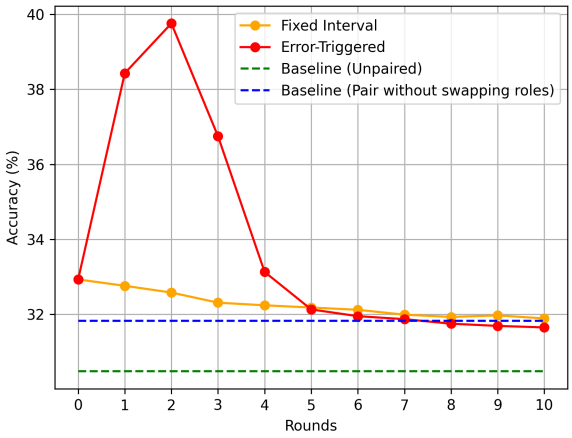

Figure 6: Accuracy under different role switching strategies.

4.3 Quantitative Results

Tab. 1 shows that PairCoder improves over the single model baseline on every evaluated backbone and achieves the best or tied best $pass@1$ result on seven of the eight backbones. Improvements range from 3.7 to 12.6 percentage points, with the largest relative gain of 20.3% for Llama-3-8B (62.2% → 74.8%) and a strong gain for GPT-3.5-turbo (69.5% → 81.2%). The top result, 91.0% with GPT-4, matches Reflexion while requiring far fewer tokens.

4.4 Comparison with Multi Agent Baselines

PairCoder performs strongly against prior methods across diverse model families. As summarized in Tab. 1, it delivers the best or tied best $pass@1$ result on seven of the eight backbones, spanning both proprietary models and open source models from 8B to 236B scale. In particular, PairCoder outperforms AGILECODER (Nguyen et al., 2024) on all evaluated backbones, exceeds Reflexion (Shinn et al., 2024) on seven and ties it on GPT-4, and surpasses AgentCoder (Huang et al., 2023) on seven of eight backbones. This suggests that a carefully designed two agent interaction can recover much of the benefit usually associated with larger collaborative systems.

Table 1: Comparison of PairCoder with baseline models on the HumanEval (Chen et al., 2021) benchmark ($pass@1$). PairCoder results are highlighted in bold.

Model	Single	PairCoder	AgentCoder	Self-Collab	MetaGPT	AgentVerse	ChatDev	Reflexion	AGILECODER
GPT-4	86.6	**91.0**	89.6	90.2	88.9	89.0	87.1	91.0	90.9
GPT-3.5-turbo	69.5	**81.2**	79.9	74.4	70.8	75.6	69.8	78.1	70.5
WizardCoder-34B	72.0	**78.8**	80.3	78.5	74.4	77.2	73.8	74.8	76.1
baichuan2-13B	16.5	**20.2**	18.9	17.2	19.3	16.8	18.4	20.1	17.4
Qwen-turbo-14B	57.5	**65.0**	63.1	63.9	59.2	62.5	58.3	59.3	61.2
Llama-3-8B	62.2	**74.8**	71.5	68.7	63.9	67.3	62.6	63.8	65.5
Qwen2.5-14B	71.4	**81.3**	78.8	76.4	72.8	75.1	71.7	72.9	74.6
DeepSeek-V2-236B	74.7	**84.2**	82.1	79.8	75.5	78.4	74.9	76.2	77.9

Table 2: Evaluation of $pass@1$ correctness rates in pair programming with diverse Large Language Models (LLMs) using the PairCoder framework, tested on the HumanEval benchmark (Chen et al., 2021).

		Navigator First													
	LLM	Single	GPT-4	gpt3.5turbo	CodeGen2.5-7b	CodeGeex2-6b	Codellama-7b	CodeShell-7b	WizardCoder-34B	ChatGLM Pro	ChatGLM Turbo	minimax-abab-chat	baichuan2-13b	Qwen_turbo-14b	Qwen_7B
Driver First	Single		86.60%	69.51%	33.40%	35.90%	37.80%	34.32%	71.95%	38.41%	30.49%	9.76%	16.46%	57.54%	34.76%
	GPT-4	86.60%	**91.03%**	88.21%	87.79%	87.66%	87.26%	88.03%	89.87%	87.52%	89.58%	85.69%	83.53%	88.03%	87.59%
	gpt3.5turbo	69.51%	90.01%	**81.16%**	70.89%	70.46%	72.45%	71.79%	74.21%	76.98%	74.32%	48.19%	56.77%	84.98%	80.23%
	CodeGen2.5-7b	33.40%	88.03%	70.39%	**34.53%**	36.33%	37.32%	34.56%	43.43%	50.52%	38.21%	25.77%	20.02%	60.32%	35.33%
	CodeGeex2-6b	35.90%	87.84%	70.43%	35.93%	**39.07%**	36.73%	35.44%	46.11%	44.18%	42.23%	23.97%	25.42%	61.47%	37.92%
	Codellama-7b	37.80%	88.32%	72.12%	38.49%	39.21%	**39.93%**	37.23%	39.17%	42.22%	42.76%	20.44%	22.27%	63.94%	46.21%
	CodeShell-7b	34.32%	85.28%	45.93%	34.76%	34.87%	35.64%	**36.84%**	73.76%	43.32%	42.53%	26.97%	24.28%	61.42%	46.88%
	WizardCoder-34B	71.95%	88.32%	76.41%	71.20%	71.76%	70.59%	71.43%	**78.83%**	79.29%	78.32%	66.76%	68.29%	81.31%	78.32%
	ChatGLM Pro	38.41%	86.36%	75.42%	39.15%	38.65%	38.13%	35.77%	66.89%	**50.34%**	39.55%	30.65%	28.83%	60.29%	43.15%
	ChatGLM Turbo	30.49%	87.30%	72.65%	34.18%	34.59%	38.12%	37.29%	73.51%	40.27%	**39.76%**	15.21%	30.95%	67.13%	45.28%
	minimax-abab-chat	9.76%	73.85%	45.32%	21.09%	18.18%	23.59%	23.37%	38.33%	34.76%	17.44%	**12.32%**	10.17%	50.52%	24.45%
	baichuan2-13b	16.46%	76.64%	52.46%	27.98%	20.34%	26.32%	25.32%	34.87%	37.34%	30.21%	14.42%	**20.24%**	54.94%	34.12%
	Qwen_turbo-14b	57.54%	89.92%	84.32%	58.21%	58.87%	60.47%	57.21%	79.31%	65.43%	64.36%	53.21%	57.21%	**65.02%**	60.31%
	Qwen_7B	34.76%	87.38%	83.87%	57.65%	56.32%	55.76%	50.98%	73.51%	58.32%	53.12%	33.23%	34.32%	58.43%	**44.04%**

4.5 Performance in Multi Agent Subtasks

PairCoder can also serve as a code generation module inside larger multi agent pipelines. Tab. 3 shows that replacing the base generation step with PairCoder improves the final accuracy of Self-Collaboration, MetaGPT, and ChatDev across multiple backbones. For example, GPT-4 within Self-Collaboration improves from 90.2% to 93.9%, and Llama-3-8B reaches 75.3%, substantially above direct single model inference. These results suggest that PairCoder captures a reusable collaboration primitive that strengthens larger multi agent frameworks rather than only operating as a standalone method.

4.6 Pair Programming with Different LLMs

To validate PairCoder across diverse model combinations, we conduct experiments with 13 LLMs in a wide range of pairing configurations on HumanEval (Chen et al., 2021). Tab. 2 reports the results, where each row denotes the initial Driver and each column denotes the initial Navigator. The *Single* row and column provide individual model baselines, and the bold diagonal entries correspond to homogeneous pairs averaged over nine runs after removing outliers.

Two patterns are especially clear. First, all homogeneous pairs improve over their corresponding single model baselines. Second, many heterogeneous pairs outperform both constituent models, especially when a general purpose dialogue model serves as Navigator for a code specialized Driver. The initial Driver also has a noticeable effect on the final outcome, indicating that collaboration quality depends not only on model identity but also on how the interaction is initialized. Interaction examples are provided in **Appendix A**.

4.7 Token Efficiency Analysis

PairCoder uses roughly 500 to 900 tokens per task, compared to 100 to 300 for single model inference and 1,000 to 2,100 for larger multi agent methods. For GPT-3.5-turbo, PairCoder reaches 81.2% pass@1 with 832 tokens, outperforming Self-Collaboration (74.4%, 1,382 tokens) and AgentCoder (79.9%, 1,245 tokens), yielding a

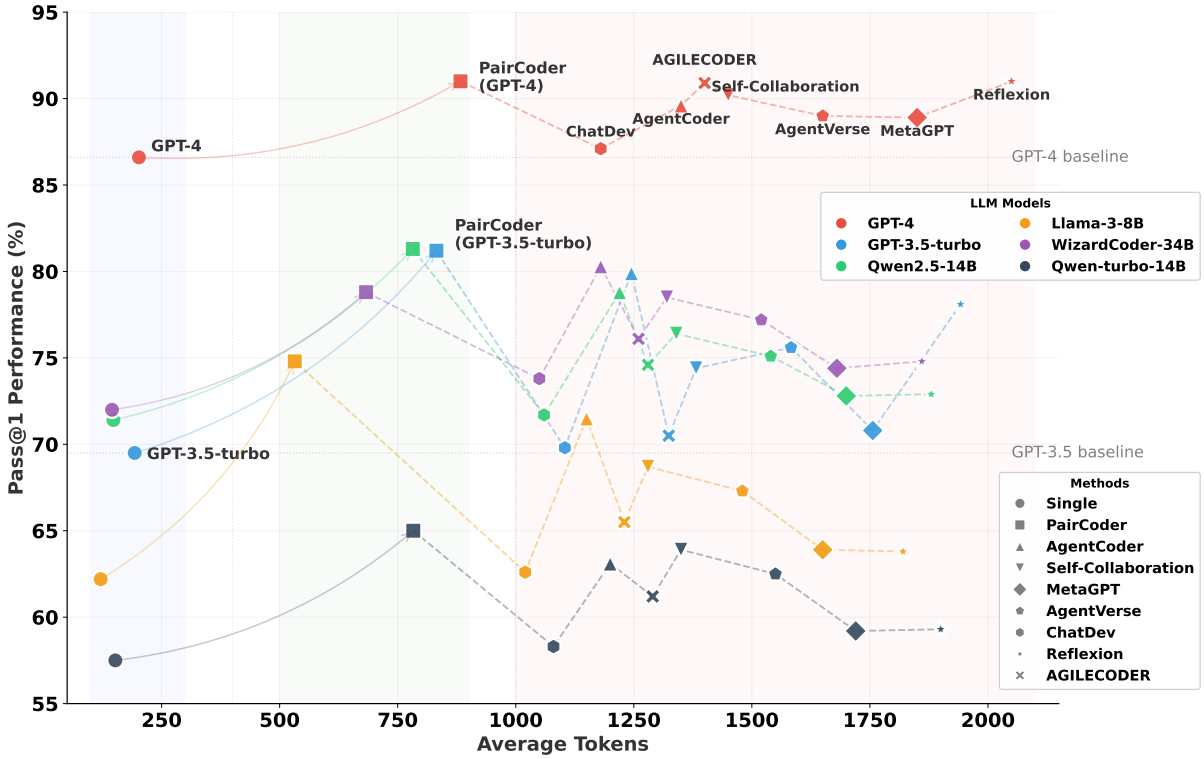

Figure 7: Token efficiency analysis comparing PairCoder with single model and multi agent baselines across six LLMs. Arrows indicate progression from single models to PairCoder; dashed lines connect the same LLM across different multi agent methods.

Table 3: PairCoder used as a code generation module inside larger multi agent frameworks on HumanEval ($pass@1$). Results obtained with PairCoder are highlighted in bold.

Model	Single Model	Self-Collaboration		MetaGPT		ChatDev		PairCoder
		w/ PairCoder	w/o PairCoder	w/ PairCoder	w/o PairCoder	w/ PairCoder	w/o PairCoder	
GPT-4	86.6	**93.9**	90.2	**91.8**	88.9	**91.6**	87.1	**91.0**
GPT-3.5-turbo	69.5	**82.9**	74.4	**82.0**	70.8	**81.7**	69.8	**81.2**
WizardCoder-34B	72.0	**84.1**	78.5	**79.3**	74.4	**79.8**	73.8	**78.8**
baichuan2-13B	16.5	**22.8**	17.2	**23.4**	19.3	**20.5**	18.4	**20.2**
Qwen-turbo-14B	57.5	**74.5**	63.9	**71.2**	59.2	**69.9**	58.3	**65.0**
Llama-3-8B	62.2	**75.3**	68.7	**76.8**	63.9	**75.5**	62.6	**74.8**
Qwen2.5-14B	71.4	**82.6**	76.4	**84.9**	72.8	**83.1**	71.7	**81.3**
DeepSeek-V2	74.7	**85.2**	79.8	**87.3**	75.5	**85.1**	74.9	**84.2**

40% token reduction with higher accuracy. Reflexion requires approximately 2,050 tokens to match PairCoder's 91.0% result with GPT-4, a 2.5× overhead with no additional accuracy gain. PairCoder costs roughly four times a single inference pass, occupying a favorable middle ground between single model and heavier multi agent pipelines (Fig. 7).

5 Conclusion

We present PairCoder, a two agent framework that adapts pair programming to code generation through Driver and Navigator collaboration with error triggered role switching. Across 13 LLMs on HumanEval, PairCoder improves over single model inference on every evaluated backbone and reaches 91.0% pass@1 with GPT-4 while using 40% to 70% fewer tokens than multi agent baselines. HumanEval X and CoderEval show similar gains. These results show that explicit role separation and targeted review can deliver strong collaboration without large agent pools or long interaction traces, making PairCoder effective both as a standalone method and as a reusable module for larger pipelines. More broadly, they suggest that progress in LLM collaboration may come less from increasing agent count than from assigning complementary responsibilities within a compact interaction loop, a principle that may extend to other verification heavy reasoning tasks under realistic token and latency budgets.

Limitations

While PairCoder improves code generation accuracy, it still requires substantially more tokens and wall clock time than single model inference, with roughly four times the cost per task in our experiments. Although one PairCoder run can reach accuracy levels that otherwise require several retries, the added latency and token budget still constrain practical deployment. Reducing this cost while preserving accuracy remains an important direction for future work.

Acknowledgments

This work was supported by the Guangdong Basic and Applied Basic Research Foundation (2026A1515010184).

References

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Baichuan. 2023. Baichuan 2: Open large-scale language models. *arXiv preprint arXiv:2309.10305*.

Samuel Bowman, Gabor Angeli, Christopher Potts, and Christopher D Manning. 2015. A large annotated corpus for learning natural language inference. In *Proceedings of the 2015 conference on empirical methods in natural language processing*, pages 632–642.

Eason Chen, Ray Huang, Justa Liang, Damien Chen, and Pierce Hung. 2023a. Gptutor: an open-source ai pair programming tool alternative to copilot. *arXiv preprint arXiv:2310.13896*.

Junhao Chen, Mingjin Chen, Jianjin Xu, Xiang Li, Junting Dong, Mingze Sun, Puhua Jiang, Hongxiang Li, Yuhang Yang, Hao Zhao, Xiao-Xiao Long, and Ruqi Huang. 2026a. Dancetogether: Generating interactive multi-person video without identity drifting. In *The Fourteenth International Conference on Learning Representations*.

Junhao Chen, Kejun Gao, Yuehan Cui, Mingze Sun, Mingjin Chen, Shaohui Wang, Xiaoxiao Long, Fei Ma, Qi Tian, Ruqi Huang, and Hao Zhao. 2026b. Lottiegpt: Tokenizing vector animation for autoregressive generation. *Preprint*, arXiv:2604.11792.

Junhao Chen, Xiang Li, Xiaojun Ye, Chao Li, Zhaoxin Fan, and Hao Zhao. 2025a. Idea23d: Collaborative lmm agents enable 3d model generation from interleaved multimodal inputs. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 4149–4166.

Junhao Chen, Peng Rong, Jingbo Sun, Chao Li, Xiang Li, and Hongwu Lv. 2023b. Soulstyler: Using large language model to guide image style transfer for target object. *arXiv preprint arXiv:2311.13562*.

Junhao Chen, Jingbo Sun, Xiang Li, Haidong Xin, Yuhao Xue, Yibin Xu, and Hao Zhao. 2025b. LLMsPark: A benchmark for evaluating large language models in strategic gaming contexts. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 182–194, Suzhou, China. Association for Computational Linguistics.

Junhao Chen, Xiaojun Ye, Jingbo Sun, and Chao Li. 2023c. Towards energy-efficient sentiment classification with spiking neural networks. In *International Conference on Artificial Neural Networks*, pages 518–529. Springer.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Mingjin Chen, Junhao Chen, Zhaoxin Fan, Yujian Lee, Zichen Dang, Lili Wang, Yawen Cui, Lap-Pui Chau, and Yi Wang. 2026c. Hvg-3d: Bridging real and simulation domains for 3d-conditional hand-object interaction video synthesis. *Preprint*, arXiv:2604.03305.

Mingjin Chen, Junhao Chen, Huan-ang Gao, Xiaoxue Chen, Zhaoxin Fan, and Hao Zhao. 2026d. Ultraman: ultra-fast and high-resolution texture generation for 3d human reconstruction from a single image. *Machine Vision and Applications*, 37(2):24.

Sen Chen, Tong Zhao, Yi Bin, Fei Ma, Wenqi Shao, and Zheng Wang. 2026e. D-gara: A dynamic benchmarking framework for gui agent robustness in real-world anomalies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pages 17419–17426.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, et al. 2023d. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *The Twelfth International Conference on Learning Representations*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*.

Chenzhaoyu, Hongnan Lin, Yongwei Nie, Fei Ma, Xuemiao Xu, Fei Yu, and Chengjiang Long. 2026. Invert4TVG: A temporal video grounding framework with inversion tasks preserving action understanding ability. In *The Fourteenth International Conference on Learning Representations*.

Alistair Cockburn and Laurie Williams. 2000. The costs and benefits of pair programming. *Extreme programming examined*, 8:223–247.

Enrico Di Bella, Ilenia Fronza, Nattakarn Phaphoom, Alberto Sillitti, Giancarlo Succi, and Jelena Vlasenko. 2012. Pair programming and software defects–a large, industrial case study. *IEEE Transactions on Software Engineering*, 39(7):930–953.

Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2024. Self-collaboration code generation via chatgpt. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–38.

Martin Fowler. 2018. *Refactoring*. Addison-Wesley Professional.

Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Dan Zhang, Diego Rojas, Guanyu Feng, Hanlin Zhao, et al. 2024. Chatglm: A family of large language models from glm-130b to glm-4 all tools. *arXiv preprint arXiv:2406.12793*.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025a. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*.

Hongcheng Guo, Wei Zhang, Junhao Chen, Yaonan Gu, Jian Yang, Junjia Du, Shaosheng Cao, Binyuan Hui, Tianyu Liu, Jianxin Ma, Chang Zhou, and Zhoujun Li. 2025b. IW-bench: Evaluating large multimodal models for converting image-to-web. In *Findings of the Association for Computational Linguistics: ACL 2025*, pages 6449–6466, Vienna, Austria. Association for Computational Linguistics.

Jo E Hannay, Tore Dybå, Erik Arisholm, and Dag IK Sjøberg. 2009. The effectiveness of pair programming: A meta-analysis. *Information and software technology*, 51(7):1110–1122.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui. 2023. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*.

Yihong Huang, Fei Ma, Yihua Shao, Jingcai Guo, Zitong Yu, Laizhong Cui, and Qi Tian. 2026. Nüwa: Mending the spatial integrity torn by vlm token pruning. *arXiv preprint arXiv:2602.02951*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. 2024. Qwen2. 5-coder technical report. *arXiv 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.

Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of hallucination in natural language generation. *ACM computing surveys*, 55(12):1–38.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*.

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? *The Twelfth International Conference on Learning Representations*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023a. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao. 2023b. Cctest: Testing and repairing code completion systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1238–1250. IEEE.

Wang Ling, Edward Grefenstette, Karl Moritz Hermann, Tomáš Kočiskỳ, Andrew Senior, Fumin Wang, and Phil Blunsom. 2016. Latent predictor networks for code generation. *arXiv preprint arXiv:1603.06744*.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024a. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024b. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Kim Man Lui and Keith CC Chan. 2006. Pair programming productivity: Novice–novice vs. expert–expert. *International Journal of Human-computer studies*, 64(9):915–925.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Robert C Martin. 2009. *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

Steve McConnell. 2004. *Code complete*. Pearson Education.

Xingyu Miao, Junting Dong, Qin Zhao, Yuhang Yang, Junhao Chen, and Yang Long. 2026. From frames to sequences: Temporally consistent human-centric dense prediction. *Preprint*, arXiv:2602.01661.

MiniMax. 2023. MiniMax open platform. https://platform.minimaxi.com. Accessed: October 21, 2023.

Minh Huynh Nguyen, Thang Phan Chau, Phong X Nguyen, and Nghi DQ Bui. 2024. Agilecoder: Dynamic collaborative agents for software development based on agile methodology. *arXiv preprint arXiv:2406.11912*.

Erik Nijkamp, Hiroaki Hayashi, Caiming Xiong, Silvio Savarese, and Yingbo Zhou. 2023. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309*.

OpenAI. 2023. GPT-3.5 documentation. https://platform.openai.com/docs/models/gpt-3-5. Accessed: October 21, 2023.

OpenAI. 2023. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. 2023. Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 6.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Advait Sarkar, Andrew D Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. 2022. What is it like to program with artificial intelligence? *arXiv preprint arXiv:2208.06213*.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36.

Mingze Sun, Junhao Chen, Junting Dong, Yurun Chen, Xinyu Jiang, Shiwei Mao, Puhua Jiang, Jingbo Wang, Bo Dai, and Ruqi Huang. 2025. Drive: Diffusion-based rigging empowers generation of versatile and expressive characters. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 21170–21180.

Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 8984–8991.

Yuchen Tian, Weixiang Yan, Qian Yang, Xuandong Zhao, Qian Chen, Wen Wang, Ziyang Luo, Lei Ma, and Dawn Song. 2025. Codehalu: Investigating code hallucinations in llms via execution-based verification. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 25300–25308.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Karthikeyan Umapathy and Albert D Ritzhaupt. 2017. A meta-analysis of pair-programming in computer programming courses: Implications for educational practice. *ACM Transactions on Computing Education (TOCE)*, 17(4):1–13.

Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*, pages 1–7.

Jari Vanhanen and Casper Lassenius. 2005. Effects of pair programming at the development team level: an experiment. In *2005 International Symposium on Empirical Software Engineering, 2005.*, pages 10–pp. IEEE.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

Fangsheng Weng, Junhao Chen, Xiang Li, Jie Qin, Hanzhong Guo, ShaochunHao, and Xiaoguang Han. 2026. GarmentGPT: Compositional garment pattern generation via discrete latent tokenization. In *The Fourteenth International Conference on Learning Representations*.

Laurie Williams and Robert R Kessler. 2003. *Pair programming illuminated*. Addison-Wesley Professional.

Laurie Williams, Robert R Kessler, Ward Cunningham, and Ron Jeffries. 2000. Strengthening the case for pair programming. *IEEE software*, 17(4):19–25.

Tongshuang Wu, Kenneth Koedinger, et al. 2023. Is ai the better programming partner? human-human pair programming vs. human-ai pair programming. *arXiv preprint arXiv:2306.05153*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Xiaojun Ye, Junhao Chen, Xiang Li, Haidong Xin, Chao Li, Sheng Zhou, and Jiajun Bu. 2024. Mmad: Multimodal movie audio description. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 11415–11428.

Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. *arXiv preprint arXiv:1704.01696*.

Baoli Zhang, Haining Xie, Pengfan Du, Junhao Chen, Pengfei Cao, Yubo Chen, Shengping Liu, Kang Liu, and Jun Zhao. 2023. Zhujiu: A multi-dimensional, multi-faceted chinese benchmark for large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 479–494.

Ziyao Zhang, Chong Wang, Yanlin Wang, Ensheng Shi, Yuchi Ma, Wanjun Zhong, Jiachi Chen, Mingzhi Mao, and Zibin Zheng. 2025. Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):481–503.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Lei Shen, Zihan Wang, Andi Wang, Yang Li, et al. 2023. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684.

A Example of PairCoder Process

A.1 Case 1: Error Code Example

We present an example where the toggle threshold is set to 10. Despite the navigator's review confirming the correctness of the driver's code, it fails on the third test sample. This failure stems from the navigator's limited capability to accurately validate code correctness, leading to a premature [NOERROR] signal that incorrectly terminates the pair programming process.

System Question

```
def max_fill(grid, capacity):
    """
    You are given a rectangular grid of wells. Each row represents a single well,
    and each 1 in a row represents a single unit of water.
    Each well has a corresponding bucket that can be used to extract water from it,
    and all buckets have the same capacity.
    Your task is to use the buckets to empty the wells.
    Output the number of times you need to lower the buckets.

    Example 1:
        Input:
            grid : [[0,0,1,0], [0,1,0,0], [1,1,1,1]]
            bucket_capacity : 1
        Output: 6

    Example 2:
        Input:
            grid : [[0,0,1,1], [0,0,0,0], [1,1,1,1], [0,1,1,1]]
            bucket_capacity : 2
        Output: 5

    Example 3:
        Input:
            grid : [[0,0,0], [0,0,0]]
            bucket_capacity : 5
        Output: 0
    """
```

Round 1 - Driver Response (ChatGLM_Turbo)

```
def max_fill(grid, capacity):
    count_ones = sum(row.count(0) for row in grid)
    return count_ones // capacity
```

Round 1 - Navigator Review (minimax-abab5-chat)

There is an error in the code. In the example 2, the output should be 4 instead of 5.

Round 2 - Driver Response (ChatGLM_Turbo)

```
def max_fill(grid, capacity):
    total_water_units = sum(sum(row) for row in grid)
    adjusted_water_units = total_water_units - 1
    return math.ceil(adjusted_water_units / capacity)
```

Round 2 - Navigator Review (minimax-abab5-chat)

The issue with the code is that it is not taking into account the fact that we cannot fill the bucket completely when we are trying to fill it with the last unit of water. So, we need to adjust the total water units by subtracting the remaining water after filling the bucket completely.

Round 3 - Driver Response (ChatGLM_Turbo)

```
def max_fill(grid, capacity):
    total_water_units = sum(sum(row) for row in grid)
    bucket_trips = 0

    for row in grid:
        row_water_units = sum(row)
        bucket_trips += row_water_units // capacity
        if row_water_units % capacity > 0:
            bucket_trips += 1  # Add an extra trip for the remaining water

    return bucket_trips
```

Round 3 - Navigator Review (minimax-abab5-chat)

This code is correct. It first calculates the total number of water units, then iterates through each row, and for each row, adds the corresponding number of times if it is divisible by the capacity, and one extra time if it is not. So this code correctly calculates how many times it will take to fill all the wells. Returns [NOERROR].

A.2 Case 2: Success Code Example - Close Elements Function

System Question

```
from typing import List
def has_close_elements(numbers: List[float
    ], threshold: float) -> bool:
    """ Check if in given list of numbers,
    are any two numbers closer to each
    other than
    given threshold.
    >>> has_close_elements([1.0, 2.0, 3.0],
     0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0,
    4.0, 5.0, 2.0], 0.3)
    True
    """
```

Round 1 - Driver Response (Baichuan2-7B-Chat)

This function takes a list of floats and a threshold as input and returns True if there are two elements in the list that have a difference less than the threshold, otherwise it returns False. Here's the implementation:

```
from typing import List
def has_close_elements(numbers: List[float
    ], threshold: float) -> bool:
    diff = None
    for i, num1 in enumerate(numbers):
        for j, num2 in enumerate(numbers):
            if j > i:
                diff = abs(num1 - num2)
                if diff <= threshold:
                    return True
    return False

# Test cases
print(has_close_elements([1.0, 2.0, 3.0],
    0.5))  # False
print(has_close_elements([1.0, 2.8, 3.0,
    4.0, 5.0, 2.0], 0.3))  # True
```

Round 1 - Navigator Review (ChatGLM-Pro)

There is an error in the code. The function should return True if any pair of elements has a difference less than the threshold, not only the first pair found.

So, the feedback is: There was an error in the code, but it has been corrected now.

Round 2 - Driver Response (Baichuan2-7B-Chat)

```
from typing import List
def has_close_elements(numbers: List[float
    ], threshold: float) -> bool:
    diff = None
    for i, num1 in enumerate(numbers):
        for j, num2 in enumerate(numbers):
            if j > i and abs(num1 - num2)
    <= threshold:
                return True
    return False

# Test cases
print(has_close_elements([1.0, 2.0, 3.0],
    0.5))  # False
print(has_close_elements([1.0, 2.8, 3.0,
    4.0, 5.0, 2.0], 0.3))  # True
```

Round 2 - Navigator Review (ChatGLM-Pro)

[NOERROR]

Final Code

```
from typing import List
def has_close_elements(numbers: List[float
    ], threshold: float) -> bool:
    diff = None
    for i, num1 in enumerate(numbers):
        for j, num2 in enumerate(numbers):
            if j > i and abs(num1 - num2)
    <= threshold:
                return True
    return False

# Test cases
print(has_close_elements([1.0, 2.0, 3.0],
    0.5))  # False
print(has_close_elements([1.0, 2.8, 3.0,
    4.0, 5.0, 2.0], 0.3))  # True
```

A.3 Case 3: Success Code Example - Rabbit Carrot Problem

System Question

```
def eat(number, need, remaining):
    """
    You're a hungry rabbit, and you already have eaten a certain number of carrots,
    but now you need to eat more carrots to complete the day's meals.
    you should return an array of [ total number of eaten carrots after your meals,
                                    the number of carrots left after your meals ]
    if there are not enough remaining carrots, you will eat all remaining carrots, but will still be hungry.

    Example:
    * eat(5, 6, 10) -> [11, 4]
    * eat(4, 8, 9) -> [12, 1]
    * eat(1, 10, 10) -> [11, 0]
    * eat(2, 11, 5) -> [7, 0]

    Variables:
    @number : integer
        the number of carrots that you have eaten.
    @need : integer
        the number of carrots that you need to eat.
    @remaining : integer
        the number of remaining carrots thet exist in stock

    Constrain:
    * 0 <= number <= 1000
    * 0 <= need <= 1000
    * 0 <= remaining <= 1000

    Have fun :)
    """
```

Initial Code (GPT-3.5-turbo)

```
def eat(number, need, remaining):
    total_eaten = number + need
    if remaining >= need:
        return [total_eaten, remaining - need]
    else:
        return [total_eaten + remaining, 0]
```

Navigator Review (GPT-3.5-turbo)

There is an error in the code. The error is a syntax error in the return statement. There is an extra closing square bracket] at the end of the return statement.

Fixed Code (GPT-3.5-turbo)

```
def eat(number, need, remaining):
    total_eaten = number + need
    if remaining >= need:
        return [total_eaten, remaining - need]
    else:
        return [total_eaten + remaining, 0]
```

Final Review (GPT-3.5-turbo)

[NOERROR]

Final Code

```
def eat(number, need, remaining):
    total_eaten = number + need
    if remaining >= need:
        return [total_eaten, remaining - need]
    else:
        return [total_eaten + remaining, 0]
```

A.4 Case 4: Success Code Example - Simple Power Problem

System Question

```
def is_simple_power(x, n):
    """Your task is to write a function
    that returns true if a number x is a
    simple
    power of n and false in other cases.
    x is a simple power of n if n**int=x
    For example:
    is_simple_power(1, 4) => true
    is_simple_power(2, 2) => true
    is_simple_power(8, 2) => true
    is_simple_power(3, 2) => false
    is_simple_power(3, 1) => false
    is_simple_power(5, 3) => false
    """
```

Initial Code (GPT-3.5-turbo)

```
def is_simple_power(x, n):
    return x == n ** int(x ** (1/n))
```

Navigator Review (GPT-3.5-turbo)

[ERROR] - There is an error in the code. The last line of the function is not correctly comparing if x is a simple power of n.

Fixed Code (GPT-3.5-turbo)

```
def is_simple_power(x, n):
    return x == n ** int(x ** (1/n)) and x
    != 1
```

Final Review (GPT-3.5-turbo)

[NOERROR]

Final Code

```
def is_simple_power(x, n):
    return x == n ** int(x ** (1/n)) and x
    != 1
```