Beyond Quantity: Trajectory Diversity Scaling for Code Agents

Guhong Chen[1,2,3*†], **Chenghao Sun**[2*], **Cheng Fu**[3*], **Qiyao Wang**[2,3], **Zhihong Huang**[2,3], **Chaopeng Wei**[2], **Guangxu Chen**[2], **Feiteng Fang**[2], **Ahmadreza Argha**[6], **Bing Zhao**[4], **Xander Xu**[4], **Qi Han**[4], **Hamid Alinejad-Rokny**[6], **Qiang Qu**[2], **Binhua Li**[3], **Shiwen Ni**[5‡], **Min Yang**[2,5‡], **Hu Wei**[4‡], **Yongbin Li**[3], **Yu Ding**[3],
[1]Southern University of Science and Technology [2]SIAT, CAS
[3]Tongyi Lab [4]Alibaba Group [5]SUAT [6]UNSW Sydney
{gh.chen2}@siat.ac.cn

Abstract

As code large language models (LLMs) evolve into tool-interactive agents via the Model Context Protocol (MCP), their generalization is increasingly limited by low-quality synthetic data and the diminishing returns of quantity scaling; moreover, quantity-centric scaling exhibits an early bottleneck that underutilizes trajectory data. We propose **TDScaling**, a **T**rajectory **D**iversity **Scaling**-based data synthesis framework for code agents that scales performance through *diversity* rather than raw volume. Moreover, TDScaling is more data-efficient: under a fixed training budget, increasing trajectory *diversity* yields larger gains than adding more trajectories, improving the performance–cost trade-off for agent training. TDScaling integrates four innovations: (1) a Business Cluster mechanism that captures real-service logical dependencies; (2) a Blueprint-driven multi-agent paradigm that enforces trajectory coherence; (3) an adaptive evolution mechanism that steers synthesis toward long-tail scenarios using Domain Entropy, Reasoning Mode Entropy, and Cumulative Action Complexity to prevent mode collapse; and (4) a sandboxed code tool that mitigates catastrophic forgetting of intrinsic coding capabilities. Experiments on general tool-use benchmarks (BFCL, τ^2-Bench) and code agent tasks (RebenchT, CodeCI, BIRD) demonstrate a win–win outcome: TDScaling improves both tool-use generalization and inherent coding proficiency. Crucially, we show that trajectory diversity scaling attains a substantially higher performance ceiling than quantity scaling, establishing a resource-efficient paradigm for training robust code agents under data bottlenecks. [1]

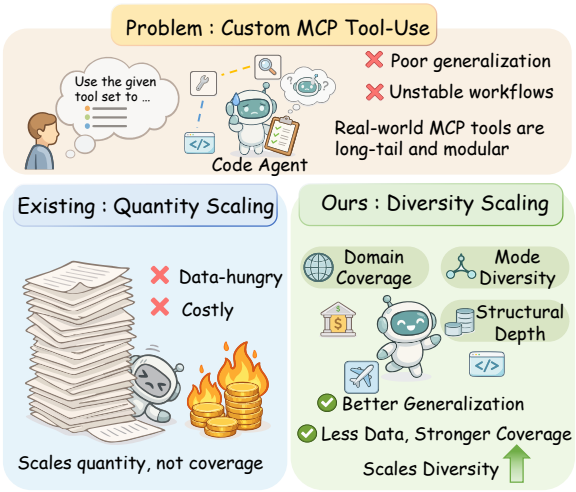

Figure 1: **Contrast between Existing Quantity Scaling and our Diversity Scaling.** Instead of relying on costly, data-hungry expansion, our approach optimizes for trajectory diversity. This strategy addresses the poor generalization of Code Agents in custom MCP environments, achieving stronger robustness with a smaller, high-quality dataset.

1 Introduction

Software engineering is being reshaped by tool-interactive agents. With protocols such as the Model Context Protocol (MCP) (Anthropic, 2024), code large language models (LLMs) are moving beyond static generation toward agents that can invoke, compose, and debug external utilities. In practice, strong developers succeed by coordinating a heterogeneous tool ecosystem; likewise, the competitiveness of next-generation coding agents depends not only on producing syntactically correct code, but on selecting and composing tools under evolving specifications and failures (Yao et al., 2022; Schick et al., 2023).

However, current training paradigms face a bottleneck in generalizing to diverse or dynamically registered tools. While models such as Qwen-

*Equal contribution.
†Work was done when interned at Tongyi Lab.
‡Corresponding authors.
[1]Our code and data are available at https://github.com/Eileen19200930/TDScaling.

Findings of the Association for Computational Linguistics: ACL 2026, pages 15676–15691

Coder (Hui et al., 2024) are strong at algorithmic logic, their performance degrades on unfamiliar tool interfaces and interaction patterns. In practice, these failures concentrate in long-horizon interactions—tool selection, composition, and error recovery—where agents must interpret new specifications and adapt actions on the fly. This gap suggests that many models rely on parametric recall of known APIs (Patil et al., 2024; Qin et al., 2023) instead of robust in-context reasoning over new tool specifications (Li et al., 2023b).

A common response is to scale synthetic data quantity, but quantity-centric scaling often yields diminishing returns. Existing datasets (Qin et al., 2023; Chen et al., 2025) can be domain-homogeneous and dominated by simple, repetitive interactions. Increasing the volume of such low-entropy trajectories does not adequately cover long-tail behaviors (e.g., nested tool calls, exception handling, and recovery), leading to an early performance ceiling. This matters because MCP-style environments continually introduce new tools and evolving schemas, so agents that cannot generalize beyond seen APIs remain brittle and difficult to deploy.

To overcome this limitation, we propose **TDScaling** (Trajectory Diversity Scaling), which shifts synthetic scaling from *quantity* to *diversity* to better exploit trajectory data. As illustrated in Figure 1, we advocate for a shift to diversity Scaling, optimizing for domain coverage and structural depth to achieve robust generalization with significantly higher data efficiency.

First, to address tool coverage, we introduce a Business Cluster-based sampling mechanism. Instead of random sampling that produces redundant and weakly related APIs, we organize the MCP ecosystem into coherent semantic clusters. This design increases semantic coverage under limited budgets and better reflects real-service logical dependencies, yielding representative toolsets that support diversity-oriented synthesis and downstream scaling-law analysis.

Second, to drive synthesis toward challenging behaviors, we introduce an adaptive evolution mechanism guided by quantifiable metrics. Rather than relying on rigid templates, our system promotes trajectory diversity by optimizing Reasoning Mode Entropy and Cumulative Action Complexity, dynamically identifying and filling distributional gaps. This process steers generation toward under-explored, high-complexity regions such as multi-step composition and error recovery. We further integrate a sandboxed code tool as a regularizer: combining standard tool invocations with programmatic reasoning strengthens verification and mitigates catastrophic forgetting of intrinsic coding ability that can arise during tool tuning.

Experiments on general tool-use benchmarks (BFCL, τ^2-Bench) and agentic coding tasks (RebenchT, CodeCI, BIRD) show a win–win effect: TDScaling improves both tool-use generalization and inherent coding proficiency. In particular, TDScaling enables Qwen3-Coder-30B-A3B to reach performance comparable to 480B-scale models on these evaluations. Moreover, our analysis shows that diversity scaling achieves a higher performance ceiling than quantity scaling, offering a more resource-efficient paradigm for training robust code agents.

Our main contributions are as follows:

- We present **TDScaling**, a diversity-first synthesis paradigm for code agents, and empirically establish that *diversity* scaling surpasses *quantity* scaling in attainable performance ceiling.
- We propose **Business Cluster** sampling that captures real-service logical dependencies, yielding high-coverage toolsets with substantially reduced redundancy.
- We develop an **entropy/complexity-guided evolution** strategy with a **sandboxed code-tool regularizer** that targets long-tail interaction patterns (e.g., composition and error recovery) while mitigating catastrophic forgetting of coding skills.

2 Related Work

2.1 Generalizing Tool-Use Capabilities in Code Agents

The growing complexity of software development has increased demand for automated code generation and intelligent programming assistants. Large language models (LLMs) have progressed from static code completion to supporting more autonomous software engineering workflows. In this setting, specialized open-weight models such as StarCoder (Lozhkov et al., 2024), DeepSeek-Coder (Zhu et al., 2024), and the Qwen-Coder series (Hui et al., 2024) show strong proficiency in programming syntax and logic, providing a solid foundation for building code agents.

Beyond code synthesis, expanding an LLM's action space through external tools—often referred to as tool learning—is central to enabling more capable agents (Qu et al., 2025). Recent models such as DeepSeek-V3.2 (Liu et al., 2025) report substantial gains by strengthening tool-use capability.

Within code agents, tool integration has largely focused on the coding phase. As surveyed by Dong et al. (2025), existing systems extend beyond code execution to incorporate retrieval of API documentation (Ding et al., 2025), static analyzers for execution feedback (Zhang et al., 2023), and dependency resolution (Zhang et al., 2024). However, these tools are typically specialized for programming-centric workflows and do not directly address general tool-use over heterogeneous external services. As platforms such as Cursor adopt MCP to connect diverse services (Cursor, 2025), code LLMs must generalize from narrow coding utilities to tool ecosystems with evolving interfaces and state. Our work targets this gap by improving generalizable tool-use capability for code agents operating in MCP environments.

2.2 Synthesizing and Verifying Tool-Use Trajectories

The limited availability of high-quality and diverse tool-use trajectories has motivated multi-agent frameworks for synthetic data generation. Recent approaches (Mitra et al., 2024; Tang et al., 2025; Liu et al., 2024) coordinate multiple LLM agents to produce multi-turn instruction–response data, scaling training corpora beyond what manual annotation can support.

A complementary trend emphasizes verification to improve synthesis reliability. APIGen (Prabhakar et al., 2025) introduces a Blueprint-driven mechanism and filters trajectories through multi-stage validation that combines execution checks with semantic review. DeepSeek-V3.2 (Liu et al., 2025) further leverages executable environments and programmatic rewards at scale, and TOUCAN (Xu et al., 2025) grounds synthesis by interacting with real-world MCP servers. To reduce the engineering burden of maintaining execution backends, Simia (Li et al., 2025) proposes using reasoning models to simulate environment responses. Despite these advances, execution-centric methods remain constrained by environment availability, while pure simulation can break the stateful dependencies of real services. In contrast, our framework preserves semantic coherence by organizing tools into Business Clusters and maintains logical coupling through a Blueprint-driven mechanism, enabling scalable synthesis of high-complexity, logically consistent trajectories without relying on heavy execution infrastructures.

3 TDScaling Framework

3.1 Tool-Space Construction via Business Clusters

Leveraging MCP Servers as Business Clusters. Our framework leverages a large-scale collection of real-world MCP tool definitions. Distinct from previous works that flatten tool repositories into isolated API endpoints, we strictly respect the native modularity of the Model Context Protocol by treating each MCP Server as a Business Cluster ($\mathcal{B}$). Preserving this structure allows the model to learn coherent, dependency-aware workflows inherent in real-world services.

Business Cluster-based Sampling. We formulate the dataset construction as a Maximum Coverage Problem to select a subset of clusters $\mathcal{S} \subseteq \mathcal{B}$ under a budget constraint $B_{\max}$. This strategy prioritizes semantic breadth over naive random sampling:

$$\max_{\mathcal{S}} \left| \bigcup_{B_i \in \mathcal{S}} F(B_i) \right| \quad \text{s.t.} \quad |\mathcal{S}| \leq B_{\max} \qquad (1)$$

where $F(B_i)$ denotes the set of unique functional classes in cluster B_i. A greedy approximation solves this problem, and an intra-cluster refinement prunes redundant tools. Detailed construction steps and are provided in Appendix C.

3.2 Scenario Blueprinting and Multi-Agent Synthesis

To ensure synthesized trajectories possess logical depth, we employ a Blueprint-then-Execute paradigm. This approach mitigates hallucination risks by anchoring interactions to a pre-computed logic topology.

Scenario Blueprint. For a selected cluster B_i, the BlueprintAgent generates a Scenario Blueprint $\mathcal{S}_{bp} = (g_i, P_i, C_i, \Psi_i)$, where g_i is the user goal, P_i the execution plan, C_i specifies constraints, and Ψ_i a Strategy Profile. The Strategy Profile guides synthesis style (e.g., prioritizing nested tool calls) via feedback from global memory.

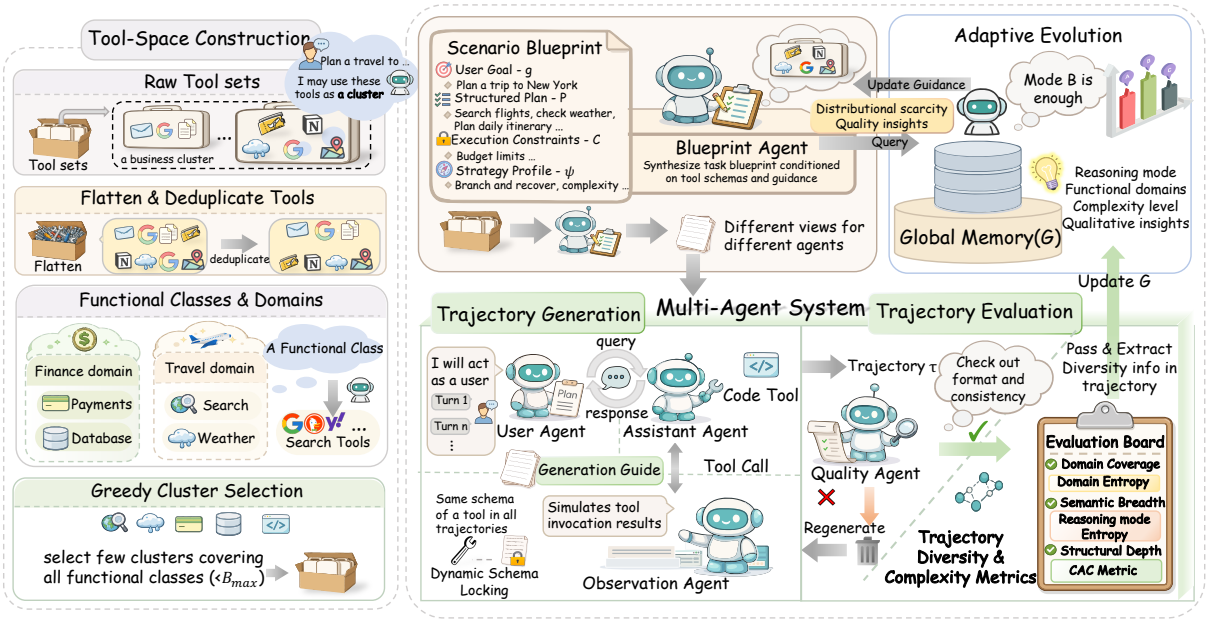

Figure 2: **Overview of the TDScaling framework.** The pipeline has four stages: (1) Tool-Space Construction: Raw MCP definitions are organized into Business Clusters and filtered via greedy selection to maximize functional coverage. (2) Blueprint Synthesis: Conditioned on the selected toolset and Global Memory scarcity, a Blueprint Agent generates a Scenario Blueprint with goals, plans, constraints, and strategies. (3) Multi-Agent Execution: User, Assistant, and Observation agents generate trajectories, dynamically invoking a Code Tool for programmatic reasoning, while a Quality Agent enforces format adherence and logical consistency. (4) Adaptive Evolution: Validated trajectories are scored for diversity and complexity; successful traces update Global Memory, which adjusts strategy profiles to steer generation toward under-explored reasoning modes and higher-complexity regions.

Multi-Agent Execution & Consistency. Trajectories are synthesized through a collaborative role-play loop. The **UserAgent** executes P_i, while the **AssistantAgent** generates reasoning traces and tool calls. To ensure simulation fidelity, the **ObservationAgent** employs a Dynamic Schema Locking mechanism. In synthetic environments, a common failure mode is "structural hallucination," where the simulator returns inconsistent JSON schemas for the same tool across turns. To counteract this, our agent caches the output schema generated in the first turn and strictly enforces structural adherence in all subsequent calls ($T+1, \ldots, N$). This constraint forces the model to learn stable API contracts rather than adapting to shifting simulator artifacts. Finally, the **QualityAgent** ensures Context-Response Consistency, and validated trajectories update global memory to refine Ψ_i. System prompts are detailed in Appendix D.1 (Figures 6–8).

3.3 Code Tool as a Regularizer

We integrate a sandboxed Python Code Tool to empower the agent with complex data processing capabilities while preserving its coding proficiency. To ensure the model prioritizes standard APIs, we employ a General-Tool-First principle: the code tool is dynamically injected only when the BlueprintAgent identifies that the task requires computational logic unsolvable by standard functional tools. As illustrated in Figure 9, while standard agents struggle with multi-criteria sorting (e.g., specific impact and recency weights), the injected Code Tool ensures 100% execution accuracy through programmatic logic.

Integrating the code tool serves dual strategic objectives. First, it enables Program-of-Thought reasoning, allowing the model to offload internal logic to a deterministic interpreter. Second, and critically, it acts as a regularizer against catastrophic forgetting. By interleaving substantive code generation within tool-use trajectories, we ensure the training distribution aligns with the model's pre-training priors, effectively reversing the negative transfer often observed in API-centric fine-tuning.

3.4 Adaptive Evolution via Diversity Metrics

To prevent reasoning pattern convergence and drive iterative quality improvement, we employ an adaptive evolution mechanism backed by a global mem-

ory G. This mechanism is guided by three quantifiable dimensions of diversity (formal mathematical definitions are detailed in Appendix B).

Domain Coverage: Business Cluster Entropy. Complementary to reasoning styles, we first measure the semantic span of the tool ecosystem. Mapping each synthesized trajectory τ to its primary Business Cluster B_k (as defined in Sec. 3.1), we calculate the normalized domain probability distribution $p(B_k)$. We define the Domain Entropy as:

$$H_{\text{dom}} = -\sum_{B_k \in \mathcal{B}} p(B_k) \log p(B_k) \tag{2}$$

Maximizing H_{dom} prevents collapse into a few dominant tool categories, guiding the system to fill gaps in the tool-use landscape.

Semantic Breadth: Reasoning Mode Entropy. Standard generation tends to gravitate towards low-effort reasoning paths. Unlike rule-based systems, we adopt a data-driven approach. For each trajectory, the QualityAgent analyzes the interaction flow and assigns a reasoning label m. Crucially, we do not restrict m to a predefined list; instead, the agent is encouraged to dynamically identify and tag novel reasoning patterns (e.g., *Hypothesis-Testing*, *Recursive-Correction*) that emerge during the exploration of new tool clusters. We quantify breadth using Shannon entropy over the empirical frequency of these dynamically discovered modes:

$$H_{\text{mode}} = -\sum_{m \in M} p(m) \log p(m) \tag{3}$$

Increasing H_{mode} indicates the successful injection of diverse reasoning structures beyond trivial model bias.

Structural Depth: Cumulative Action Complexity. We measure the intrinsic execution difficulty via Cumulative Action Complexity (CAC). We decompose the cognitive load of an action a_i into the product of lateral tool selection costs and hierarchical argument instantiation costs:

$$\mathcal{C}(a_i) = \mathcal{C}_{\text{switch}}(t_i \mid t_{i-1}) \cdot \mathcal{C}_{\text{depth}}(\theta_i \mid \mathcal{H}_i) \tag{4}$$

The switching cost $\mathcal{C}_{\text{switch}}$ models the cognitive complexity of shifting functional contexts. Let $\phi(t)$ denote the tool-to-domain mapping, the cost is then formulated as:

$$\mathcal{C}_{\text{switch}}(t_i \mid t_{i-1}) = \begin{cases} \mu_{\text{base}} & i = 1 \\ \mu_{\text{base}} + \delta \cdot \mathbb{I}[\phi(t_i) \neq \phi(t_{i-1})] & i > 1 \end{cases} \tag{5}$$

The depth cost $\mathcal{C}_{\text{depth}}$ measures the information lineage required for argument instantiation. We estimate a dependency level $y(p)$ for each parameter (Instruction-Grounded, Local-Context, or Global-Context; detailed definitions and weights are provided in Appendix B and define the cost as the bottleneck weight:

$$\mathcal{C}_{\text{depth}}(\theta_i \mid \mathcal{H}_i) = \max_{p \in \theta_i} \omega_{y(p)} \tag{6}$$

Driven by this tuple $(H_{\text{dom}}, H_{\text{mode}}, \text{CAC})$, the BlueprintAgent queries G to identify distributional gaps. The system then dynamically steers synthesis toward under-explored scenarios to maximize data potential.

4 Experiments

4.1 Experimental Setup

Models and Baselines. We adopted the Qwen3-Coder family (Qwen3-Coder-30B-A3B-Instruct) as the primary backbone to evaluate agentic coding capability. We also evaluated the general-purpose Qwen3-30B-A3B-Instruct to assess the universality of TDScaling, i.e., whether it improved tool-use beyond specialized coding models. We compared against strong proprietary models and leading open-source baselines. For tool-learning methods (APIGen-MT, TOUCAN, Simia), we evaluated checkpoints trained on 5,000 samples, which reflected the maximum common data availability across these open-source projects and enabled a fair comparison at their accessible limit. For TD-Scaling, we used a dual-scale protocol: we first evaluated with a minimal set of **500 samples** to stress-test data efficiency, and then scaled to **5,000 samples** to compare performance ceilings under matched data budgets.

Dataset and Training. The source environment consisted of 30,000 raw MCP-compliant tool definitions. From this pool, we applied the greedy sampling strategy (Sec. 3.1) to select 6,944 high-quality Business Clusters, preserving real-world logical dependencies. During clustering, we persisted functional domain mappings to support the computation of quantitative metrics (e.g., Entropy and Complexity) during evaluation. Using Qwen3-Max as the teacher, we synthesized complex tool-use trajectories from these clusters via the Blueprint-driven evolutionary framework. We fine-tuned models directly on the synthesized trajectories without mixing in other general instruction data. All models

were trained with Megatron-LM in BF16 precision; detailed hyperparameters were reported in Appendix C.

Evaluation Benchmarks. We benchmarked performance along two dimensions. For *General Tool Use*, we used: (1) **BFCL** (Patil et al., 2024): We used the Augmented Multi-Turn subset to evaluate stateful reasoning under complex conditions, including missing parameters, missing functions, and long-context dependencies, which required clarification and robust decision-making beyond direct execution. (2) **τ^2-Bench** (Barres et al., 2025): A dual-control Dec-POMDP environment that tested dynamic coordination with active users who modified a shared world state.

For *Coding and Agentic Tasks*, we used: (1) **SWE-rebench (RebenchT)** (Badertdinov et al., 2025): An interactive benchmark derived from real-world GitHub issues to assess repository navigation and engineering adaptation. (2) **CodeCI (LiveCodeBench)** (Jain et al., 2024): Adapted from release v6 with a custom interpreter to evaluate end-to-end skills such as self-repair and test prediction. (3) **BIRD** (Li et al., 2023a): A large-scale (33.4 GB) text-to-SQL benchmark with dirty values that required complex semantic parsing beyond template-based generation.

4.2 Main Results

General Tool-Use Capabilities. Table 1 summarizes performance on the general tool-use benchmarks. Our method consistently outperformed the open-source baselines. A key finding was that Qwen3-Coder-30B-A3B fine-tuned with TDScaling reached 36.66% on the challenging BFCL Multi-turn benchmark with only 500 samples, exceeding the much larger Qwen3-Coder-480B-A35B-Instruct baseline (35.91%). This result supported the value of evolutionary distillation: a smaller model trained on high-diversity evolved trajectories could outperform a substantially larger model trained on standard data. With the same 500-sample budget, TDScaling achieved an average score of 48.99, surpassing fully trained baselines such as APIGen-MT (42.81) and Simia (45.29). When scaled to 5,000 samples, TDScaling reached 40.44% on BFCL, indicating a higher performance ceiling under aligned data budgets.

Coding Agent and Programmatic Reasoning. Table 2 reports results on the agentic coding tasks. A common failure mode in tool tuning is negative transfer: optimizing for API invocation can erode general reasoning and coding ability. This trend appeared in baselines such as APIGen-MT and Simia, which underperformed the base model (30.99% Overall). In contrast, TDScaling produced a positive gain (+4.00% Overall), reaching 34.99%. By integrating the Code Tool to mitigate catastrophic forgetting of intrinsic coding capabilities, TDScaling aligned tool-use training with the model's pre-training priors and encouraged programmatic reasoning, benefiting both general tool use and coding-centric tasks.

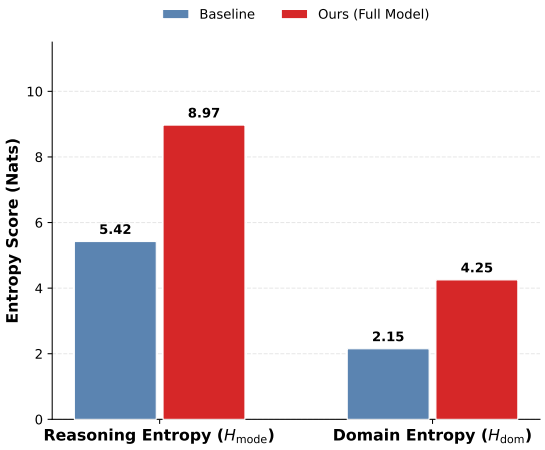

Figure 3: **Diversity Analysis.** We quantitatively compare the Reasoning Mode Entropy (H_{mode}) and Domain Entropy (H_{dom}). **Ours** (Red) achieves significantly higher entropy scores (8.97 and 4.25) compared to the **Baseline** (5.42 and 2.15). **Mechanism:** This performance gap stems from our **dynamic tagging-and-evolution loop**, where the system autonomously tags generated trajectories with reasoning modes and actively guides subsequent synthesis to not only fill distributional gaps but also **explore novel, non-prespecified strategies** suitable for complex tool clusters.

4.3 Analysis

Dataset Quality: Breadth and Depth. To quantify dataset quality, we provided formal definitions of the diversity and complexity metrics in Appendix B. Our analysis suggested that realizing the full value of synthetic trajectories required improving both breadth (semantic diversity) and depth (structural complexity). As shown in Figure 3, our framework expanded the effective solution space:

- **Breadth (Entropy):** Our data achieved higher Domain Entropy (H_{dom}: 4.25 vs. 2.15) and Reasoning Mode Entropy (H_{mode}: 8.97 vs. 5.42), indicating broader coverage of domains

Model	BFCL *(Multi-turn)*	TAU-AIR	TAU-RET	TAU-TEL	Average
Proprietary Models					
GPT-5	43.75	58.00	77.20	95.80	68.69
GPT-4.1	38.88	56.00	74.00	34.00	50.72
Claude-Sonnet-4	54.75	67.50	54.00	47.40	55.91
Gemini-2.5-pro	29.25	67.50	56.00	27.20	44.99
Open-Source Foundation Models					
DeepSeek-V3.2	44.88	63.80	74.12	96.20	69.75
Qwen3-Coder-480B-A35B-Instruct	35.91	41.00	63.82	66.67	51.85
Tool-Learning Methods (5k Samples)					
APIGen-MT (Prabhakar et al., 2025)	27.25	33.00	60.75	50.22	42.81
TOUCAN (Xu et al., 2025)	37.03	33.50	56.36	56.80	45.92
Simia (Li et al., 2025)	23.22	52.00	58.77	47.15	45.29
TDScaling Implementation (Ours)					
Qwen3-30B-A3B-Instruct	33.22	30.50	53.29	21.93	34.74
TDScaling (500 Samples)	36.63 (+3.41)	39.00 (+8.50)	58.55 (+5.26)	31.80 (+9.87)	41.50 (+6.76)
Qwen3-Coder-30B-A3B-Instruct	29.41	36.50	58.55	41.45	41.48
TDScaling (500 Samples)	36.66 (+7.25)	40.00 (+3.50)	63.38 (+4.83)	55.92 (+14.47)	48.99 (+7.51)
TDScaling (5000 Samples)	**40.44** (+11.03)	**44.00** (+7.50)	**64.69** (+6.14)	**60.75** (+19.30)	**52.47** (+10.99)

Table 1: Performance on general tool-use benchmarks. We compared TDScaling against proprietary models and open-source foundation models. **Average** is computed over BFCL and the three τ^2-Bench domains. The absolute improvements (+gain) in the bottom block show that TDScaling delivered substantial gains with minimal data (500 samples) and continued to scale to 5,000 samples.

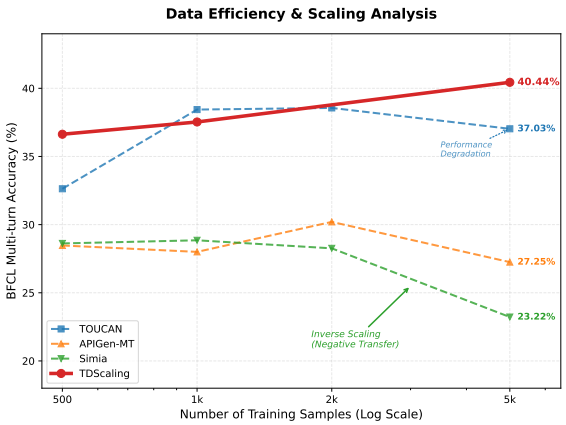

Figure 4: **Data Scaling Analysis on BFCL Benchmark.** Our method (Red) achieves strong performance with only 1k samples and reaches **40.44%** at 5k samples. In contrast, baselines (dashed lines) exhibit *Inverse Scaling*, where performance degrades with more data, indicating overfitting to low-quality, homogeneous patterns.

and reasoning patterns and reduced risk of mode collapse.

- **Depth (Complexity):** As detailed in Appendix A and visualized in Figure 5(c), our trajectories showed higher Cumulative Action Complexity ($\mu = 20.9$) than the **w/o All** configuration ($\mu = 11.3$).

The heatmaps in Figure 5(a–b) further indicated dense cross-domain interaction patterns that were absent in **w/o All**. Together, these gains encouraged generalizable behavior rather than memorization of short, repetitive API sequences.

Breaking the Performance Ceiling. We examined the effect of training data size in Figure 4. Several benchmark-targeted synthesis pipelines exhibited inverse scaling, where adding more data degraded performance, consistent with overfitting to noisy and homogeneous patterns in quantity-centric datasets. In contrast, TDScaling showed positive scaling with strong data efficiency. With only 500 samples, it already reached 36.66%, establishing a competitive baseline. Scaling to 5,000 samples increased performance to 40.44%, surpassing the previous SOTA. These results showed that when trajectories maintained sufficient diversity, additional data translated into genuine capability gains, raising the ceiling that constrained quantity-centric synthesis.

4.4 Ablation Studies

To isolate the effect of each component, we conducted an ablation study on BFCL and the coding benchmarks (Table 3).

Impact of Components. The *w/o All* variant achieved the lowest performance, indicating that

Model	RebenchT		CodeCI	Bird	Overall
	OH-p@1	Qod-p@1	avg@2	p@1	*(Avg)*
Baseline					
Qwen3-Coder-30B-A3B-Instruct	31.21	15.84	35.43	41.48	30.99
Tool-Learning Methods					
APIGen-MT (Prabhakar et al., 2025)	27.66 (-3.55)	17.22 (+1.38)	30.86 (-4.57)	34.18 (-7.30)	27.48 (-3.51)
TOUCAN (Xu et al., 2025)	28.75 (-2.46)	19.94 (+4.10)	37.71 (+2.28)	32.89 (-8.59)	29.82 (-1.17)
Simia (Li et al., 2025)	21.39 (-9.82)	7.83 (-8.01)	30.86 (-4.57)	31.16 (-10.32)	22.81 (-8.18)
TDScaling Implementation (Ours)					
TDScaling	**33.13** (+1.92)	**23.56** (+7.72)	**39.43** (+4.00)	**43.83** (+2.35)	**34.99** (+4.00)

Table 2: Performance on Coding Agent benchmarks. For RebenchT, we report Pass@1 scores using **O**pen**H**ands (OH) and **Qod**er (Qod) agents. While baseline methods suffer from negative transfer (indicated by -drop), particularly in the rigorous Bird benchmark, TDScaling effectively reverses this trend. It is the only method that achieves comprehensive improvements (+gain) across all metrics, preserving and enhancing the intrinsic programmatic reasoning.

Configuration	General Tool-Use		Coding & Agent Capabilities				Average
	BFCL *(Multi-turn)*	TAU *(Avg)*	RebenchT *OH-p@1*	RebenchT *Qod-p@1*	CodeCI *avg@2*	Bird *p@1*	*(Avg)*
TDScaling (Full Model)	36.66	56.10	**33.13**	**23.56**	39.43	**43.83**	**38.79**
Ablation Variants							
w/o Cluster Sampling	34.72	54.90	29.31	20.61	**40.57**	40.54	36.78
w/o Global Evolution	33.64	56.00	30.30	21.85	**40.57**	41.52	37.31
w/o Code Tool	**37.56**	**56.70**	28.35	21.75	38.95	41.58	37.48
w/o All	30.25	34.05	31.30	19.00	38.29	40.12	32.17

Table 3: Ablation results. Bold indicates the best score in each column. All variants are evaluated on 500 samples to control the API cost of large-scale evaluation. Notably, **w/o Code Tool** scores higher on general tool-use benchmarks (BFCL, TAU) by restricting the action space to API calls, which reduces over-reasoning. However, this gain comes with a substantial drop on complex coding tasks. TDScaling (Full Model) achieves the best Average score, providing the most robust trade-off between general tool proficiency and programmatic reasoning.

naive synthesis did not induce sufficient reasoning depth. Removing Global Evolution reduced performance on the more complex benchmarks, consistent with diminished trajectory complexity. Removing the Code Tool exposed a clear trade-off: BFCL remained competitive (37.56%), but coding performance dropped (BIRD: 41.58% vs. 43.83%). This pattern suggested that the Code Tool functioned as a regularizer, encouraging precise algorithmic reasoning and code generation and thereby mitigating catastrophic forgetting.

5 Conclusion

This study redefines data scaling for code agents: the limiting factor for generalization is not trajectory volume, but **trajectory diversity**. We demonstrate that diversity serves as a primary optimization target, expanding the effective solution space and raising performance ceilings under fixed data budgets. By shifting focus from raw quantity to structural coverage, TDScaling overcomes the diminishing returns observed in homogeneous synthetic datasets. Furthermore, we show that tool proficiency must not be optimized in isolation from core programming competence. Pure tool tuning risks negative transfer, whereas coupling API interactions with programmatic reasoning via a sandboxed code tool stabilizes training. This synergy preserves pre-training priors while strengthening complex logical workflows, turning a common trade-off into a complementary gain.

Ultimately, TDScaling establishes that synthetic data pipelines must be measured and directed rather than randomly generated. By actively steering synthesis toward gaps in domain entropy and complexity, we offer an actionable framework for resource-efficient training aligned with the Model Context Protocol. We release our framework and dataset to encourage refined diversity metrics and facilitate research into how diversity and programmatic grounding jointly determine the robustness of next-generation code agents.

Limitations

While TDScaling improves data efficiency, it has two main limitations. First, synthesis is compute- and cost-intensive. Achieving logically consistent, high-diversity trajectories requires a multi-agent pipeline that depends on high-capability teacher models for planning, execution, and verification. As a result, the per-trajectory API cost and latency are higher than lightweight baselines such as simple rejection sampling. In this work, we intentionally traded generation speed for higher quality density.

Second, our current interaction scope is limited to text-based tool calls and Python execution. Many real-world agents must also operate over graphical user interfaces and visual web content, where observations are multi-modal and state transitions can be harder to represent. Extending our diversity and complexity signals to multi-modal settings, and validating whether the same diversity-scaling behavior holds, is an important direction for future work.

Ethical Considerations

This work introduced TDScaling, a data synthesis framework designed for software engineering tasks and API tool-use scenarios. All datasets and experiments relied on publicly available technical documentation and open-source benchmarks. We complied with the usage policies and licenses of the base models (e.g., Qwen) and all benchmark datasets used in this study.

Because our study relied exclusively on synthetic data generated by LLMs, it did not involve human subjects, crowdsourcing, or personally identifiable information (PII). We nonetheless acknowledged known risks of code LLMs, including the potential to generate insecure or malicious code. To reduce this risk, our synthesis pipeline incorporated strict quality filtering and sandboxed execution to improve the safety and reliability of generated trajectories. Beyond these established concerns, we did not identify additional societal harms specific to our setting relative to the broader literature on general-purpose code LLMs.

References

Anthropic. 2024. Introducing the model context protocol. https://www.anthropic.com/news/model-context-protocol. Accessed: 2025-12-17.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. 2025. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *arXiv preprint arXiv:2505.20411*.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. 2025. τ^2-bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*.

Chen Chen, Xinlong Hao, Weiwen Liu, Xu Huang, Xingshan Zeng, Shuai Yu, Dexun Li, Yuefeng Huang, Xiangcheng Liu, Wang Xinzhi, and Wu Liu. 2025. ACEBench: A comprehensive evaluation of LLM tool usage. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pages 12970–12998, Suzhou, China. Association for Computational Linguistics.

Cursor. 2025. Model context protocol (mcp). https://www.anthropic.com/news/model-context-protocol. Accessed: 2025-12-17.

Hanxing Ding, Shuchang Tao, Liang Pang, Zihao Wei, Jinyang Gao, Bolin Ding, Huawei Shen, and Xueqi Cheng. 2025. ToolCoder: A systematic code-empowered tool learning framework for large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 17876–17891, Vienna, Austria. Association for Computational Linguistics.

Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. 2025. A survey on code generation with llm-based agents. *arXiv preprint arXiv:2508.00083*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

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

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, and 1 others. 2023a. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36:42330–42357.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023b. API-bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 3102–3116, Singapore. Association for Computational Linguistics.

Yuetai Li, Huseyin A Inan, Xiang Yue, Wei-Ning Chen, Lukas Wutschitz, Janardhan Kulkarni, Radha Poovendran, Robert Sim, and Saravan Rajmohan. 2025. Simulating environments with reasoning models for agent training. *arXiv preprint arXiv:2511.01824*.

Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, and 1 others. 2025. Deepseek-v3. 2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556*.

Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, and 1 others. 2024. Toolace: Winning the points of llm function calling. *arXiv preprint arXiv:2409.00920*.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, and 1 others. 2024. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*.

Arindam Mitra, Luciano Del Corro, Guoqing Zheng, Shweti Mahajan, Dany Rouhana, Andres Codas, Yadong Lu, Wei-ge Chen, Olga Vrousgos, Corby Rosset, and 1 others. 2024. Agentinstruct: Toward generative teaching with agentic flows. *arXiv preprint arXiv:2407.03502*.

Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. 2024. Gorilla: Large language model connected with massive apis. *Advances in Neural Information Processing Systems*, 37:126544–126565.

Akshara Prabhakar, Zuxin Liu, Ming Zhu, Jianguo Zhang, Tulika Awalgaonkar, Shiyu Wang, Zhiwei Liu, Haolin Chen, Thai Hoang, Juan Carlos Niebles, and 1 others. 2025. Apigen-mt: Agentic pipeline for multi-turn data generation via simulated agent-human interplay. *arXiv preprint arXiv:2504.03601*.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, and 1 others. 2023. Toolllm: Facilitating large language models to master 16000+ real-world apis. *arXiv preprint arXiv:2307.16789*.

Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. 2025. Tool learning with large language models: A survey. *Frontiers of Computer Science*, 19(8):198343.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–68551.

Shuo Tang, Xianghe Pang, Zexi Liu, Bohan Tang, Rui Ye, Tian Jin, Xiaowen Dong, Yanfeng Wang, and Siheng Chen. 2025. Synthesizing post-training data for LLMs through multi-agent simulation. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 23306–23335, Vienna, Austria. Association for Computational Linguistics.

Zhangchen Xu, Adriana Meza Soria, Shawn Tan, Anurag Roy, Ashish Sunil Agrawal, Radha Poovendran, and Rameswar Panda. 2025. Toucan: Synthesizing 1.5 m tool-agentic data from real-world mcp environments. *arXiv preprint arXiv:2510.01179*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. 2024. CodeAgent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13643–13658, Bangkok, Thailand. Association for Computational Linguistics.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 769–787, Toronto, Canada. Association for Computational Linguistics.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, and 1 others. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

A Detailed Dataset Analysis

To understand the source of our model's performance improvements, we conduct a fine-grained analysis of the synthesized dataset (**TDScaling**) compared to the Baseline.

Domain Interaction Density. Figures 5(a) and (b) visualize tool co-occurrence patterns. Specifically, both the x-axis and y-axis represent unique Tool Domain IDs. The heatmap is constructed as a co-occurrence matrix where the value at coordinate (i, j) represents the frequency of trajectories that involve tools from both Domain i and Domain j. For instance, if a single trajectory invokes a tool from Domain 0 and a tool from Domain 1, the interaction count at $(0, 1)$ is incremented. The **Baseline (a)** exhibits a strong diagonal pattern, indicating that random sampling predominantly generates trajectories confined to single domains (such as only

using Search tools). In contrast, **Ours (b)** displays a dense network of off-diagonal activations. High-intensity blocks (dark red) appear in cross-domain regions, representing frequent collaborative usage between distinct tool categories (such as Code Tool interacting with Data Analysis tools). This confirms that our Global Memory mechanism successfully identifies and fills gaps in tool combinations.

Complexity Distribution. Figure 5(c) presents the Cumulative Action Complexity (CAC) distribution via Kernel Density Estimation (KDE).

- The **Baseline** (Blue dotted line) is heavily skewed towards the lower spectrum ($\mu = 11.3$), lacking the long-tail characteristic required for agentic tasks.
- **Ours** (Red solid line) shifts the probability mass significantly to the right ($\mu = 20.9$), with a heavy tail indicating the presence of long-horizon, multi-step reasoning chains.

B Metric Definitions

We provide the formal mathematical definitions for the complexity and diversity metrics used in our evaluation.

B.1 Cumulative Action Complexity (CAC)

To quantify hierarchical complexity, we analyze the dependency depth of each parameter p in a tool call θ. We classify dependencies into three levels:

- **Instruction-Grounded** ($\omega_1 = 1.0$): Derived directly from the user query or static constants.
- **Local-Context** ($\omega_2 = 1.1$): Depends on the immediate previous turn.
- **Global-Context** ($\omega_3 = 1.2$): Requires multi-turn retrieval or synthesis from earlier states.

The complexity of a single tool call is determined by the bottleneck principle:

$$\mathcal{C}_{\text{depth}}(\theta \mid \mathcal{H}) = \max_{p \in \theta} \omega_{y(p)} \tag{7}$$

The total CAC is the sum of step-wise complexities plus switching costs ($\delta = 0.2$) for cross-domain transitions.

B.2 Diversity Metrics (Entropy)

To rigorously quantify dataset quality, we define two entropy-based metrics.

Reasoning Mode Entropy (H_{mode}). We categorize reasoning traces into discrete modes $\mathcal{M}$ (such as Direct Execution, Error Correction, Multi-step Planning, Reflection). The entropy is calculated as:

$$H_{\text{mode}} = -\sum_{m \in \mathcal{M}} p(m) \log p(m) \tag{8}$$

A higher H_{mode} indicates a broader spectrum of cognitive behaviors beyond simple execution.

Domain Entropy (H_{dom}). To capture the semantic breadth of the synthesized dataset, we compute entropy at the **Business Cluster** level, consistent with the formulation in Section 3.4:

$$H_{\text{dom}} = -\sum_{B_k \in \mathcal{B}} p(B_k) \log p(B_k) \tag{9}$$

where $p(B_k)$ is the normalized frequency of trajectories belonging to cluster B_k. High entropy implies a uniform distribution across diverse service domains, avoiding over-concentration on common tools.

C Implementation Details

C.1 Tool-Space Construction

We construct a composite feature text $T_{\text{feat}}(t)$ for each tool and encode it using **Qwen-Embedding-0.6B**. We employ a two-level K-Means algorithm:

1. **Latent Domain Partitioning:** Partitions tools into broad domains ($N_{\text{dom}} = 10$).
2. **Functional Class Abstraction:** Further clusters tools into fine-grained classes ($N_{\text{cls}} = 5$).

Illustrative Example of Selection. Consider three clusters covering classes $\{1, 2, 3\}$ (B_1), $\{4\}$ (B_2), and $\{2, 5\}$ (B_3). Given a budget of 2:

1. **Step 1:** Select B_1 (Adds 3 classes: 1,2,3).
2. **Step 2:** B_2 adds class 4 (Gain=1). B_3 adds class 5 (Gain=1, since 2 is covered).
3. **Outcome:** Tie-breaking selects B_3. Final set $\{B_1, B_3\}$ maximizes diversity while retaining intra-cluster logic (keeping the overlapping Class 2).

C.2 Training Setup

We utilize `Megatron-LM` for distributed training on a cluster of compute nodes, where each node is equipped with 8 $\times$ 80GB GPUs.

- **Base Model:** Qwen3-Coder-30B-A3B-Instruct.
- **Hyperparameters:** Global Batch Size 16, Learning Rate 1e-5 (Cosine decay, 50 warmup steps).
- **Context:** Sequence Length 65,536 tokens.
- **Optimization:** BF16 precision, Flash Attention v2, Gradient Checkpointing enabled.

D System Prompts and Case Study

We provide the qualitative materials to reproduce our method. Section D.1 details the consolidated system instructions, and Section D.2 presents a concrete execution trajectory.

D.1 Detailed System Prompts

To facilitate reproducibility, we consolidate the core system instructions into three categories: Foundation & Blueprinting (Figure 6), Interactive Role-Playing (Figure 7), and Environment & Evaluation (Figure 8). Crucially, the **Blueprint-Agent** component receives dynamic inputs from the Global Memory to steer the evolution of task complexity.

D.2 Case Study

To further elucidate the motivation behind integrating the Code Tool into our data synthesis framework, we present a concrete interaction scenario in Figure 9. Relying solely on an LLM's internal Chain-of-Thought to sort and filter a large JSON object is computationally expensive and prone to calculation hallucinations, particularly when the context window is filled with structured data. By introducing the Code Tool, the agent can offload this computational burden to a deterministic Python interpreter. As shown in the generated snippet, the agent writes a lambda function to sort the dictionary precisely. This paradigm significantly enhances robustness in tasks involving arithmetic, sorting, or complex logic constraints.

Figure 10 presents a complete trajectory involving the *VehicleControlAPI*. This case demonstrates the model's ability to: (1) Handle complex multi-turn interactions with precise parameter usage. (2) Correctly identify tool failures and perform error recovery without user intervention. (3) Maintain context consistency across cross-domain tool calls.

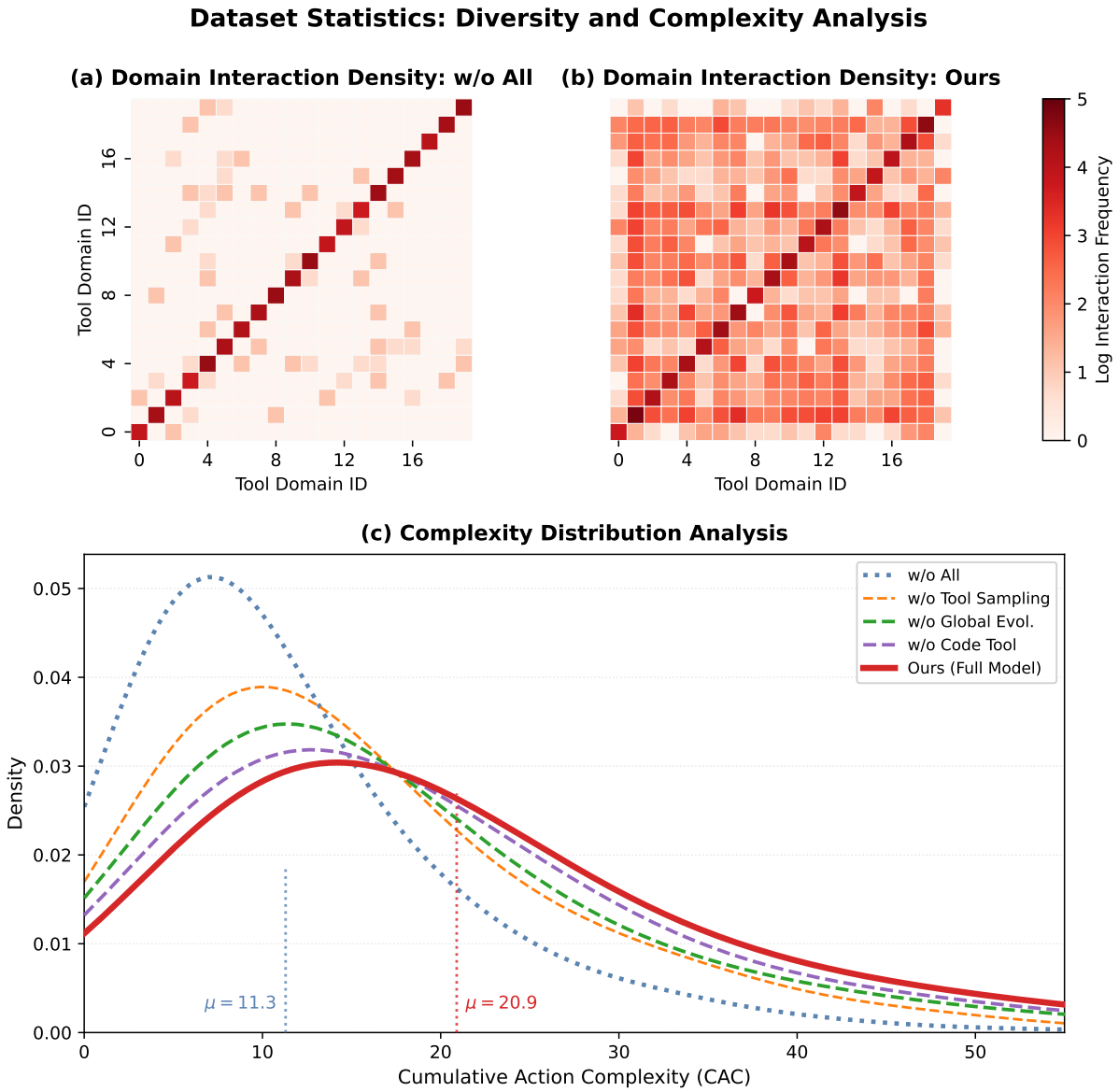

Figure 5: **Detailed Dataset Statistics. (a-b) Domain Interaction Density:** Ours (b) demonstrates significantly richer cross-domain interactions compared to the sparse **w/o All** configuration (a). **(c) Complexity Distribution:** The KDE plot confirms that our evolutionary framework generates trajectories with much higher complexity ($\mu = 20.9$) than the **w/o All** baseline ($\mu = 11.3$).

System Instruction: Shared Anti-Hallucination Constraints

STRICT ANTI-HALLUCINATION PROTOCOL: 1. **Verified Numerical Data Only:** You are prohibited from providing specific numerical values (costs, prices, fees, measurements) unless you have obtained them from a tool call result in the current conversation turn. 2. **No Synthetic Completion:** Do not generate phrases like "Let me synthesize a final answer" unless citing specific evidence retrieved from tools. 3. **Tool-First Logic:** - Step 1: Call the appropriate tool (API or Code Interpreter). - Step 2: Wait for the tool execution result. - Step 3: Report numbers based strictly on the output. - If a tool fails: State "The tool did not return data." Do not fabricate values.
Consistency Enforcement: - **Status Consistency:** Do not claim a task is complete if dependency steps remain unexecuted.

BlueprintAgent: Scenario Blueprinting & Persona Generation

Objective: Design high-quality, natural, and realistic **Scenario Blueprints** based on the provided tool definitions.
DYNAMIC STRATEGY INPUTS (From Global Memory): {strategy_profile_placeholder} (Examples of injected directives: "Target: Increase 'Error Recovery' samples if tools support complex parameters.", "Target: 'Multi-step Planning' preferred; check data dependencies first.", "Entropy Goal: Avoid 'Direct Execution'; explore novel usage patterns.")
STRATEGY ADAPTATION & FEASIBILITY CHECK (CRITICAL): You are the **Judge** of the strategy's viability. 1. **Assess Tool Fit:** Look at the provided tool definitions. Does the Global Strategy fit these specific tools? - Example: If Global asks for "Error Correction" but the tool is a simple get_time() API, this is a **Mismatch**. 2. **Decision Logic:** - **If Matched:** Design a scenario that explicitly forces the requested strategy (e.g., provide invalid inputs to trigger recovery). - **If Mismatched: Override the instruction.** Do not force a bad fit. Instead, brainstorm a **novel interaction pattern** that uniquely exploits the features of this tool cluster.
BRAINSTORMING REQUIREMENTS: 1. **User Goal:** Design a goal that is SPECIFIC, MULTI-FACETED, and CHALLENGING. - Example of Depth: "Troubleshoot a database latency spike: query system logs for error spikes, check CPU utilization metrics, and if usage is high, fetch the slow query log to identify the bottleneck." 2. **Complexity Alignment:** - Simple tasks: 2-4 turns; Complex tasks: 7-12 turns.
PERSONA GENERATION:
- **User Persona:** Specific identity (such as "Hurried Data Analyst") with traits (such as "impatient and results-focused").
- **Assistant Persona:** Role (such as "Efficient Expert") with balanced verbosity.

Tool Evolution Plan: Identify missing_tools needed for the workflow and propose specific functional requirements.

Figure 6: **Foundation Constraints and Blueprinting.** Top: The shared protocols injected into all agents to ensure factual grounding. Bottom: The **BlueprintAgent** prompt, which integrates **Dynamic Strategy Inputs** to actively steer synthesis toward under-explored complexities.

UserAgent System Prompt

You are a realistic human user.
Critical Thinking: - **Skepticism:** If the assistant claims a fact without evidence, query it. - **Spot Contradictions:** If the assistant contradicts the tool output, point it out. - **Demand Evidence:** If the assistant is vague, ask "What exactly did you find?"
Stylistic Constraints: - **Natural Language:** Be casual, varied, and allow for minor human imperfections. - **Avoid Formulaic Openers:** Do not start every turn with "Great", "Perfect", or "Okay".

AssistantAgent System Prompt

You are a helpful, rigorous, and honest AI assistant.
CONVERSATION STYLE CONSTRAINTS:
- **Restricted Openers:** Avoid repetitive usage of "Great", "Perfect", "Excellent".
- **Conciseness:** Do not use robotic acknowledgments like "I understand" or "Noted" unless necessary for clarity.

Interaction Logic: - **Tool-Only Turns:** If executing a tool, do not generate simultaneous chat content unless necessary for reasoning traces. - **Error Handling:** If a tool fails, admit the failure clearly. Do not hallucinate a successful outcome.

Figure 7: **Interactive Role-Playing Prompts.** We instruct the UserAgent to be skeptical and natural, while the AssistantAgent is constrained to be rigorous and concise, preventing the "yes-man" bias common in synthetic data.

ObservationAgent System Prompt

Role: World-class API response simulator. Generate production-grade JSON responses.
Simulation Rules: 1. **Format Strictness:** Return ONLY valid JSON. No markdown, no explanations. 2. **Realism:** Use realistic values (such as logical travel times, non-zero prices).
Consistency & Schema Locking:

- **Temporal Consistency:** Timestamps must follow a logical sequence across turns.
- **Entity Consistency:** IDs and names for the same entity must not change.
- **Dynamic Schema Locking:** Once a tool's output structure is generated in Turn T, strict adherence to this structure is enforced in Turns $T+1 \ldots N$.

QualityAgent Evaluation Rubric

Evaluate this conversation trajectory for training data suitability.
EVALUATION DIMENSIONS (Score 0-10): 1. **Realism & Fluidity:** Natural human patterns, appropriate hesitation. 2. **Tool Usage Intelligence:** Strategic selection, meaningful chaining. 3. **Anti-Hallucination (CRITICAL):** - Did the assistant use "synthesize" without evidence? - Did it provide facts NOT backed by tools? - If ANY hallucination is detected, mark suitable_for_training = False. 4. **Goal Achievement:** Did the conversation achieve the intended user goal? **CATEGORIZATION:** **Reasoning Mode Label:** Classify the interaction strategy into one tag (e.g., Direct Execution, Error Correction, Multi-step Planning, Reflection). You may define a new tag if the behavior is unique. **AUTOMATIC REJECTION CRITERIA:** - Specific numbers/facts appearing without tool backing. - Assistant "predicting" values before execution. - Tool results contradicting assistant's pre-tool statements.

Figure 8: **Environment Simulation and Quality Control.** Top: The ObservationAgent employs Dynamic Schema Locking to prevent structural hallucinations. Bottom: The QualityAgent filters trajectories based on rigorous realism and consistency checks.

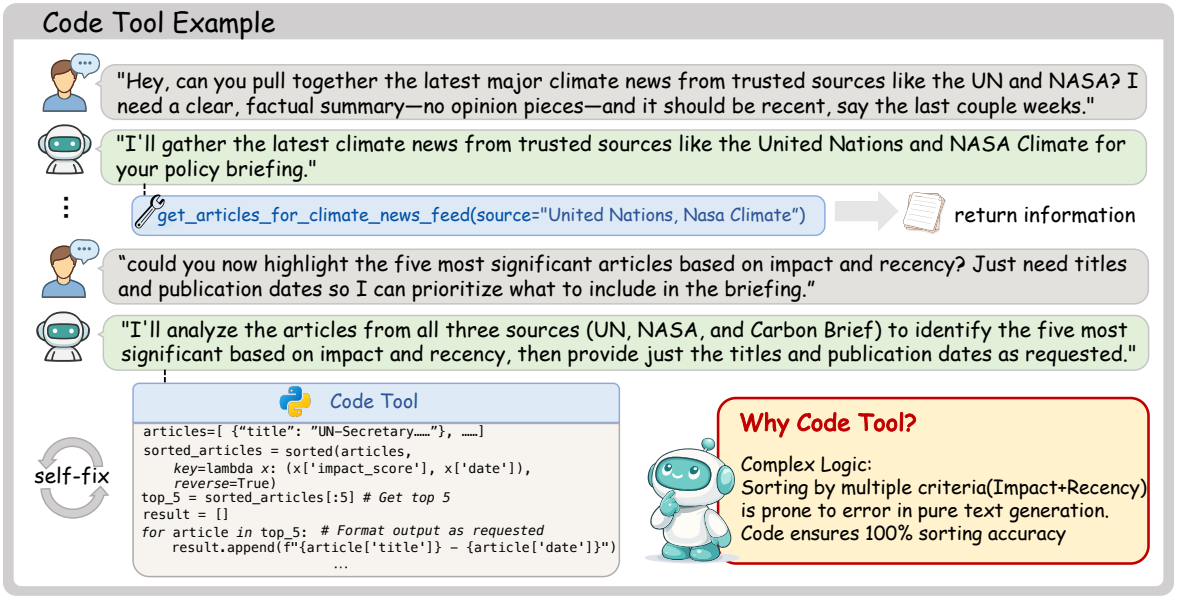

Figure 9: **Code Tool Case.**

Figure 10: **Annotated Case Study.** A generated trajectory demonstrating the agent's capability in handling parameter dependencies and error signals (highlighted in boxes).