

ToolWeave: Structured Synthesis of Complex Multi-Turn Tool-Calling Dialogues

Dinesh Khandelwal¹, Gnana Prakash Punnavajhala^{2*}, G P Shrivatsa Bhargav¹,
Gaurav Pandey¹, Sachindra Joshi¹, Hima Karanam¹, Dinesh Raghu¹

¹IBM, ²IIIT Hyderabad

¹{dikhand1, gpshri27, gpandey1, jsachind, hkaranam, diraghu1}@in.ibm.com,

²punnavajhala.prakash@research.iiit.ac.in

Abstract

Multi-turn tool calling is essential for LLMs to function as autonomous agents, yet synthesizing the training data required for these capabilities remains a fundamental challenge. Existing synthetic data generation pipelines often produce unrealistic dialogues for two reasons: they chain tools that are only superficially compatible rather than aligned with meaningful user tasks, and they generate dialogues in one shot, which often introduces arguments that were neither provided by the user nor produced by prior tool calls. These issues also lead to a severe underrepresentation of multi-step tool interactions. We introduce *ToolWeave*, a structured framework for synthesizing realistic multi-turn tool-calling dialogues. *ToolWeave* supports realistic multi-step workflows (or tool sequences) by constructing tools with built-in dependencies and filtering candidate workflows based on their alignment with user goals. It reduces parameter hallucinations through a fine-grained planning stage that explicitly tracks parameter provenance. As a result, 44.87% of conversational turns in *ToolWeave*-generated data are multi-step, and the data contains fewer hallucinations in parameters and tool names. Consequently, LLMs fine-tuned on *ToolWeave* consistently outperform those fine-tuned on prior datasets across three public benchmarks. Notably, Llama-3.1-70B fine-tuned on *ToolWeave* achieves 39.75% on BFCL-V3, compared to 23.50% when fine-tuned on SOTA *ToolFlow* data.

1 Introduction

The evolution of large language models (LLMs) into autonomous agents relies on *multi-turn tool-calling*: the ability to chain tools, elicit user information, and maintain state across complex workflows (Yao et al., 2023; Schick et al., 2023). Fine-tuning on synthetic data has become a standard approach for improving

these capabilities (Qin et al., 2024). However, generating realistic multi-turn tool-calling dialogues is not simply generating fluent conversations with tool calls. Existing synthesis pipelines, such as *ToolFlow* (Wang et al., 2025), *Magnet* (Yin et al., 2025), and *ToolDial* (Shim et al., 2025), suffer from two major limitations: semantically incoherent workflows and parameter hallucinations.

Semantically incoherent workflows: Given a set of tools, existing approaches construct a tool graph and sample sequences of connected tools (i.e., workflows) for dialogue synthesis. However, the coherence of these workflows is fundamentally limited by the quality of the underlying tool graph. In most pipelines, edges in the tool graph are inferred by matching parameter names and descriptions across tools. For instance, a pipeline may connect `create_support_ticket(issue) → ticket_id` with `cancel_movie_ticket(ticket_id)` solely because both expose a parameter named `ticket_id`. While this enforces schema-level compatibility, it does not guarantee semantic coherence.

This issue has an important downstream consequence. Because workflows are constructed based on superficial similarity rather than true data dependencies, they rarely require the output of one tool to meaningfully drive another. As a result, the synthesized dialogues seldom contain genuine *multi-step* tool interactions within a turn. This is a crucial limitation, as real world tool use tasks require composing multiple dependent operations, where intermediate results must be propagated across tools to achieve a single user objective. In our analysis (Table 2), fewer than 7% of conversational turns in data from prior pipelines contain multiple tool calls.

Parameter hallucinations: Given a candidate workflow, existing approaches typically prompt an LLM to generate the entire dialogue or plan in a single pass. This leaves the execution structure implicit. The model must infer how tool calls align with

*Work done during Gnana’s internship at IBM.

user intent, where each argument originates from, and how intermediate results are propagated. In practice, this often leads to hallucinated arguments, where parameters are neither provided by the user nor produced by preceding tool calls. As shown in Table 3, more than 50% of dialogues generated by prior approaches contain such parameter hallucinations, and models trained on this data risk inheriting the same behaviour.

These limitations stem from a common root cause: the absence of structured, verifiable intermediate representations that explicitly capture workflow dependencies and parameter provenance. **Our Approach.** To address these challenges, we introduce *ToolWeave*, a modular framework that decomposes synthesis into structured, verifiable stages. ToolWeave is designed to generate dialogues with realistic multi-step tool interactions while minimizing hallucinations.

First, instead of starting with a fixed set of tools and inferring edges in the tool graph post hoc via semantic similarity, we propose an iterative curriculum that synthesizes tools with built-in information flow between them. This yields tool graphs with meaningful data dependencies and coherent workflows by construction. We further improve workflow quality through an explicit verification step. We sample workflows, synthesize user goals for each workflow, score the compatibility between workflows and goals, and retain only high-quality pairs, thereby filtering out incoherent compositions. This increases the prevalence of multi-step tool interactions from $\sim 7\%$ in prior approaches to $\sim 45\%$.

Second, we introduce a fine-grained planning stage that explicitly models *parameter provenance*. For each dialogue, the plan specifies parameters at each step and their sources (user, context, or prior tools). This explicit structure eliminates ambiguity in execution and significantly reduces hallucinated arguments, lowering hallucination rates from $\sim 53\%$ in prior approaches to nearly 20%.

ToolWeave consistently outperforms existing approaches across three publicly available tool-calling benchmarks. The code, synthesized tools, and dialogues are publicly available.¹

2 Related Work

Fine-Tuning Datasets and Models for Tool Calling: Toolformer (Schick et al., 2023), replaced text segments with tool calls to train LLMs to pro-

duce tool calls. Several instruction-tuning datasets have been created to improve tool-calling capabilities in LLMs. Most of these datasets introduce single-turn tool-calling capabilities in models (Tang et al., 2023; Qin et al., 2024; Liu et al., 2024; Basu et al., 2024; Shi et al., 2025) using APIs extracted from sources such as RapidAPI (RapidAPI, 2026) or APIBench (Patil et al., 2024). Reinforcement learning and other tuning strategies to improve tool calling capabilities are presented by Hao et al. (2025) and Chen et al. (2025b). Tool retrieval becomes an important aspect with increasing tools and is addressed by Qin et al. (2025) and Chen et al. (2024). Recent work explores looking at creating multi-turn tool calling data using synthetic data generation (SDG) pipelines and models (Wang et al., 2025; Shim et al., 2025; Prabhakar et al., 2026; Liu et al., 2025; Yin et al., 2025; Chen et al., 2025a). The APIs and frontier models used in these pipelines make them restrictive and not licensable for commercial deployments. Random sampling of APIs followed by high-level plan generation used in these approaches often results in incoherent and poor-quality dialogues. To address these shortcomings, ToolWeave proposes an end-to-end synthetic data generation pipeline that provides fine-grained control at each stage that works equally well with frontier and small open-weight models.

Tool Calling Evaluation Benchmarks:

BFCL (Patil et al., 2025) is one of the comprehensive benchmarks aimed at evaluating various aspects of function calling including multi-turn. Several other benchmarks evaluate single-turn capabilities of the models like (Wu et al., 2024; Patil et al., 2025; Ross et al., 2025; Xu et al., 2023; Zhuang et al., 2023; Huang et al., 2024; Basu et al., 2025). Benchmarks like CONFETTI (Alkhoul et al., 2025), APIBank (Li et al., 2023), ToolHop (Ye et al., 2025) are built for evaluating multi-turn capabilities of the models. τ -bench (Yao et al., 2025) and Agentboard (Ma et al., 2024) assess agentic capabilities, including multi-turn tool use and policy adherence.

Appendix B compares ToolWeave with other SDG pipelines across key features, including code availability, API synthesis, licensing, and fine-grained control.

3 ToolWeave Architecture

ToolWeave is a modular pipeline for synthesizing realistic multi-turn, multi-step tool-calling di-

¹<https://github.com/IBM/ToolWeave>

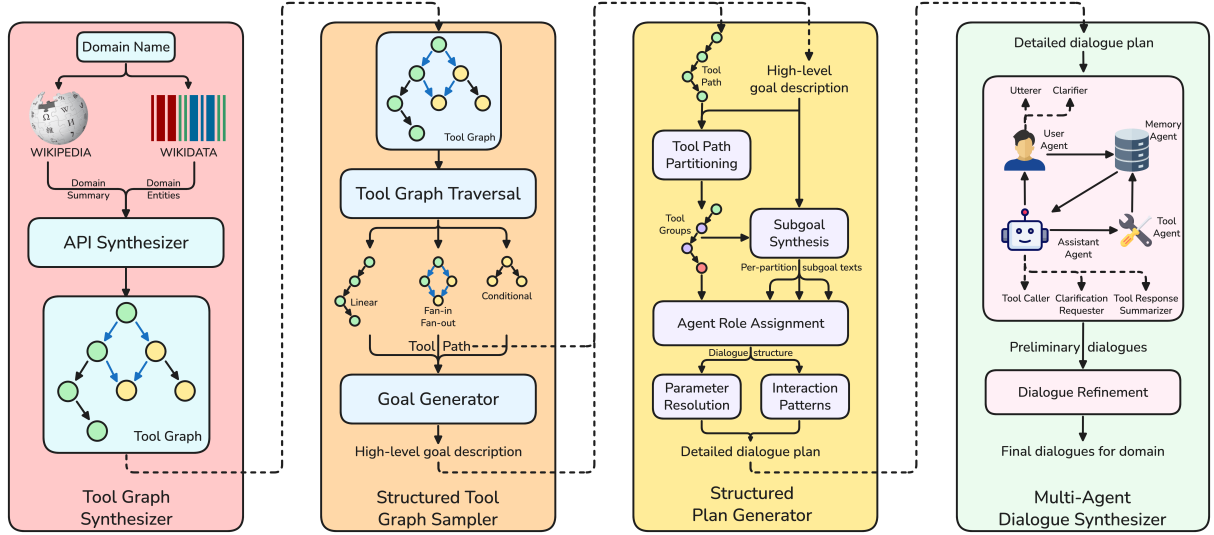


Figure 1: The modular architecture of ToolWeave. Starting with a domain name, the *Tool Graph Synthesizer* creates synthetic APIs and organizes them into an interconnected graph; the *Structured Tool Graph Sampler* extracts coherent subgraphs as goals; the *Structured Plan Generator* expands these goals into detailed dialogue plans; and the *Multi-Agent Dialogue Synthesizer* instantiates the plans into natural multi-turn dialogues.

alogues from automatically generated tools. It decomposes dialogue generation into four stages that progressively transform structured intermediate representations into natural conversations. As shown in Figure 1, ToolWeave first constructs a synthetic tool graph, then samples goal-directed workflows, converts them into fine-grained executable plans, realizes these plans as multi-agent dialogues with grounded tool calls, and finally applies post-processing to introduce linguistic variation and controlled noise. By externalizing complex reasoning into these structured stages, ToolWeave enables reliable generation of high-quality tool-calling trajectories. Figure 3 traces one tool dependency through all four stages on a running customer-support example; the complete goal, plan, tools, and dialogue for this example appear in Appendix M.

3.1 Tool Graph Synthesizer

Prior tool graph construction typically relies on either real-world APIs from marketplaces like RapidAPI (RapidAPI, 2026) or one-shot synthetic tool generation via LLMs. The RapidAPI-based approaches provide realistic tools but have two limitations. First, existing frameworks typically infer connections between tools using semantic similarity over parameter names and descriptions (Shim et al., 2025; Wang et al., 2025). Such heuristics often create incorrect edges; e.g., unrelated parameters like `get_hotel_id(.)` $\rightarrow$ `id` and `reserve_restaurant(id)` may be falsely linked

One-Shot	ToolWeave
<pre> "parameters": { "schedule_details": "string", "priority": "high medium low", "machine_id": "string" } </pre>	<pre> "parameters": { "maintenance_window": { "start_date": "date", "end_date": "date" }, "recurring": boolean, "interval_days": integer, "technician_ids": ["string"], "equipment_id": "string", "notes": "string" } </pre>

Figure 2: Input schema comparison for `generate_maintenance_schedule()`. One-shot generation yields a flat parameter list, while ToolWeave produces a richer schema with nested objects.

as compatible, leading to invalid tool compositions. Second, many tools lack clear redistribution rights, raising potential licensing concerns.

Synthetic generation approaches (Liu et al., 2025; Chen et al., 2025a) avoid licensing issues but generate all tools in a single step. As a result, the generated tools tend to cover only superficial functionality, rarely capturing deeper schema relationships, real-world business logic, or workflows across tools.

In contrast, ToolWeave’s multi-stage curriculum produces highly structured and realistic definitions. Figure 2 illustrates this contrast by comparing the input parameter schemas for a `generate_maintenance_schedule()` tool. While the one-shot baseline generates a flat, simplistic list of parameters, ToolWeave generates complex structures such as nested objects, conditional

parameter dependencies, and explicit identifiers that link to other tools. This is made possible by a curriculum-driven, multi-stage tool generation process that ToolWeave follows, described next.

1. *Seed generation*: Create a minimal set of core tools covering fundamental operations.
2. *Entity expansion*: Expand coverage using structured knowledge from Wikipedia and Wikidata. E.g., ToolWeave captures the business rule that `interval_days` is mandatory when `recurring` is true. This is inferred from Wikipedia’s maintenance page (see Figure 2).
3. *Schema enrichment*: Add depth via nested objects, enums, defaults, and optional parameters.
4. *Connection discovery*: Introduce explicit linking parameters that enable workflows across tools (e.g., `technician_id` and `equipment_id`), rather than inferring edges post-hoc via semantic similarity.

By decomposing tool synthesis into staged steps, ToolWeave constructs a domain-specific tool graph $\mathcal{G}$, where nodes represent tools and directed edges indicate that outputs of one tool may serve as inputs to another. This graph captures explicit data dependencies between tools and serves as the foundation for workflow sampling in the next stage. More details on the Tool Graph Synthesizer are provided in Appendix C.

3.2 Structured Tool Graph Sampler

Given the synthesized tool graph, the next step is to sample coherent tool workflows aligned with meaningful user goals. Prior work (Wang et al., 2025) generates tool sequences using random walks over tool graphs. While this ensures syntactic compatibility between tools, it often produces sequences that do not correspond to meaningful user tasks. ToolWeave instead enforces *tool-goal alignment* by sampling workflows that follow common structural motifs: *linear*, *fan-in/fan-out*, and *conditional*. These motifs mimic real-world tool interactions and enable the generation of goal-directed tool sequences. By constraining sampling to such motifs, ToolWeave transforms tool-sequence generation from an unconstrained graph traversal into a controlled workflow synthesis process.

Workflow motifs: ToolWeave samples workflows using common structural motifs: *linear*, *fan-in/fan-out*, and *conditional*. Linear chains form the backbone of the sampler, reflecting typical interactions where each step builds on previous results (e.g.,

search tickets → retrieve details → update status → notify user). These chains are obtained through a bounded beam search (Russell and Norvig, 2013) over the tool graph (Appendix D). Higher-order motifs capture more complex patterns. *Fan-in/fan-out* workflows aggregate multiple pieces of information before a final action (e.g., retrieving ticket details and checking user history before updating the ticket). *Conditional* workflows model branches where the next step depends on an API result (e.g., retrieve ticket details if creation succeeds, otherwise list existing tickets).

Validation and Selection: For each sampled workflow, ToolWeave synthesizes a natural-language goal and scores the pair (tools in workflow, goal) using an LLM judge. After discarding pairs below a minimum quality threshold, we apply Maximum Marginal Relevance (MMR) (Carbonell and Goldstein, 1998) to the linear workflows to prune similar goals and maximize task diversity. For *fan-in/fan-out* and *conditional* motifs, all valid pairs are retained to preserve complex interaction patterns. Detailed scoring rubrics and algorithms are provided in Appendix D.

3.3 Structured Plan Generator

Given a goal and its tool sequence, the next step is to generate a plan that guides dialogue synthesis. Broadly, plan generation strategies fall into two categories: high-level planning and turn-level planning. High-level plans (Wang et al., 2025) specify only abstract steps, leaving most work to the dialogue synthesizer. While flexible, this places a heavy burden on synthesizer and often leads to errors such as repeated interaction loops (e.g., repeated goodbyes), incorrect tool ordering (e.g., shipping an order before it is created), or hallucinated outputs produced without corresponding tool calls. In contrast, turn-level plans explicitly specify the structure of each dialogue turn. However, existing turn-level approaches (Shim et al., 2025) rely on predefined action-sequence templates that do not explicitly model parameter dependencies across tools, and rarely generate dialogues containing multi-step tool interactions within the same turn, producing such patterns in approximately 7% of conversational turns.

To address these limitations, ToolWeave shifts much of the reasoning burden from the dialogue synthesizer to an explicit planning stage, so that

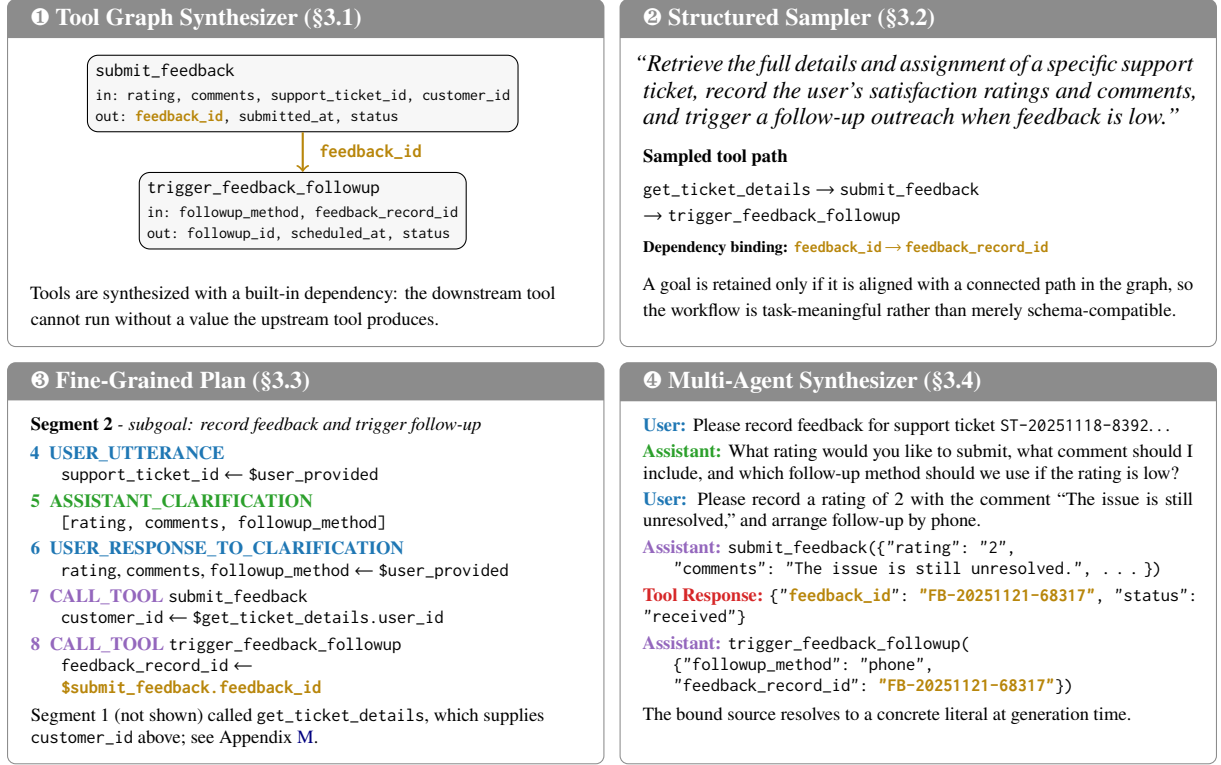


Figure 3: A cross-tool dependency traced through the four ToolWeave stages. The planner binds `trigger_feedback_followup.feedback_record_id` to its provenance source, `submit_feedback.feedback_id`, rather than to a hard-coded value. During dialogue synthesis, this binding is resolved using the upstream tool output. The full example appears in Appendix M.

the synthesizer mainly realizes the dialogue from the plan rather than performing complex reasoning. Unlike prior methods that infer dialogue structure during generation, ToolWeave computes tool grouping, parameter dataflow, and clarification requirements during planning. As a result, the planner enables richer interaction patterns, increasing the proportion of multi-step turns to 44.87%. The planner operates in three stages.

Tool Path Partitioning: Given a goal and its corresponding tool sequence (e.g., [A, B, C, D]), the planner partitions the sequence into contiguous groups (e.g., [[A], [B, C], [D]]). Each group represents a coherent sub-goal realized as a dialogue segment. Linear paths are partitioned by an LLM conditioned on the goal to identify natural subgoal boundaries. Higher-order motifs use deterministic rules to preserve dependencies between tools.

Subgoal Synthesis and Segmentation: Next, each tool group is assigned a concise natural-language subgoal, and the corresponding dialogue segment is constructed. Each segment is realized as a sequence of interaction acts including

`user_request`, `assistant_clarification`, `user_clarification_response`, `tool_invocation`, and `assistant_response`. A segment begins with a user request expressing the subgoal. Each required parameter is provided in that request with probability $1 - p_{clar}$ and withheld otherwise; withheld parameters are elicited through assistant clarification requests and user responses before the corresponding tool calls are executed and their results summarized.

Parameter Planning: Within each segment, the planner resolves tool parameters by tracing the Tool Graph. Parameters are categorized as (i) user-provided (from user utterances) and (ii) derivable (populated from prior tool outputs or dialogue history). This explicitly plans dataflow instead of relying on the dialogue synthesizer to infer dependencies during generation.

Overall, the Structured Plan Generator introduces three key design elements. Tool grouping partitions the tool path into tool groups, enabling multi-step tool interactions within a dialogue segment. Fine-grained interaction acts improve conversational coherence. Explicit parameter binding

ensures consistent data flow across turns. Together, these choices transform dialogue generation from a loosely guided language generation problem into a structured execution process, enabling reliable synthesis of complex multi-turn tool-calling dialogues. Planner algorithms are described in Appendix E.

3.4 Multi-Agent Dialogue Synthesizer and Post-Processing

Using the structured plan, ToolWeave synthesizes the dialogue using a plan-driven multi-agent framework. Each interaction act is handled by a dedicated agent, along with a tool agent and a memory agent. The memory agent maintains a persistent state containing resolved parameters, tool outputs, and other dialogue facts, ensuring consistency across turns. The tool agent generates schema-consistent responses for tool calls without requiring access to live API endpoints. The agent responsible for the current interaction act receives the dialogue history, the parameter plan, and the memory state, and produces the next utterance accordingly. The memory is then updated before executing the next step. This stateful execution ensures that dialogues faithfully follow the plan while maintaining coherent multi-turn interactions. Details are in Appendix F.

To ensure robustness, we further refine the synthesized dialogues through three post-processing steps: (i) *Linguistic Variation*: rephrasing user utterances to prevent overfitting; (ii) *Failure Injection*: with probability p_{inject} , adding a variant of the dialogue that introduces erroneous tool calls and recovery behaviors to simulate real-world API faults; and (iii) *Schema Masking*: replacing function names with generic identifiers (e.g., `func_01`) to force reasoning over rich schemas rather than surface labels. This refinement process transforms raw synthetic traces into a natural, error-robust dataset suited for generalizable tool-calling. (Detailed algorithms are in Appendix G).

4 Analysis of Data Quality

4.1 API Quality and Coverage

We evaluate the synthetic APIs generated by ToolWeave along three dimensions: *breadth*, *depth*, and *connectivity*. Breadth measures API diversity via the average number of APIs per domain and parameters per API. Depth captures schema complexity using *Complex API Use (CAU)*, the proportion of APIs containing nested objects or arrays, and the *Required Parameter Ratio (RPR)*, the fraction

Table 1: Avg. API stats per domain across synthesis models.

Metric	GPT-OSS (O)	Mistral (M)	GPT-5-mini (G)
Avg. APIs / Domain	25.25	20.85	28.35
Avg. Params / API	3.60	2.92	5.01
Complex API Use (CAU)	23.6%	19.8%	36.6%
Required Param Ratio (RPR)	65.6%	68.4%	48.9%
Interconnectivity (IC)	1.08	1.85	1.03
Longest Chain (avg.)	5.5	5.9	6.15

of parameters marked as required. Connectivity measures how APIs compose into workflows using *Interconnectivity (IC)*, which estimates direct data-flow potential between APIs, and the average longest tool chain. Formal definitions of these metrics are provided in Appendix C.7.

We generate API sets for 20 domains (Appendix C.2) using three synthesis LLMs: the open-weight GPT-OSS-120B, the commercial mistral-medium-2505, and the frontier GPT-5-mini. Table 1 reports the per-domain averages. Open-weight models produce broader schemas, while stronger models generate more complex APIs with higher nesting. Despite these stylistic differences, ToolWeave consistently produces coherent and highly connected Tool Graphs.

Comparison with Real-World APIs: Compared to the 4,474 RapidAPI schemas used in ToolDial (Shim et al., 2025), our synthetic APIs exhibit higher structural complexity (avg. 3.6-5.0 vs. 2.4 parameters; CAU up to 36.6% vs. 1.0%). We argue that this difference is beneficial: higher schema complexity introduces richer parameter structures and constraints, requiring LLMs to perform more precise parameter grounding and composition within each tool call. This intra-tool complexity provides the necessary hooks for multi-step reasoning, as reflected in Section 4.2, where our dialogues achieve up to 44.87% multi-step turns versus ~7% in baseline datasets. Moreover, complex APIs with expressive parameter spaces provide richer supervision signals for learning tool usage and parameter grounding. As shown in Section 5.2, these structural properties lead to superior multi-turn tool-calling performance. We further evaluate edge quality directly: in a manual evaluation of 100 edges presented in randomized order with the source system hidden, 74% of ToolWeave edges were judged to form semantically coherent workflows, against 12% for ToolDial (Appendix C.8).

Table 2: Statistics of synthetic dialogues grouped by synthesis pipeline.

Dataset	Min/Max/Avg. Turns	Min/Max/Avg. Tool Calls	% Total / % True Multi-step Turns
ToolWeave	1/8/2.71	1/5/3.22	44.87 / 40.67
ToolFlow	1/10/6.28	0/15/2.20	6.77 / 0.0
ToolDial	2/6/4.48	0/2/1.51	4.47 / 0.0

4.2 Dialogue Structure and Complexity

For controlled comparison, we analyze $\sim 3.2k$ ToolWeave dialogues and compare them with matched $\sim 3.2k$ subsets from the ToolFlow and ToolDial base-lines. Table 2 summarizes the structural properties of these dialogues. We report the number of conversational *turns*, total *tool calls*, and the proportion of *multi-step turns* (turns requiring multiple tool calls). We further distinguish *True multi-step turns*, where later calls directly consume outputs from earlier ones within the same turn. The results highlight a clear difference in generation behavior. ToolFlow produces long dialogues (up to 10 turns and 15 tool calls) but achieves 0.0% True multi-step turns, indicating uncontrolled sequences of shallow, independent tool calls rather than meaningful reasoning chains. ToolDial (Shim et al., 2025), despite using GPT-4o with real APIs, generates shallow conversations averaging only ~ 1.5 tool calls per dialogue and no true multi-step turns. In contrast, ToolWeave produces compact dialogues that are dense with reasoning, achieving up to 40.67% True multi-step turns. These results suggest that structured planning and tool graph reasoning enable richer multi-step interactions than existing pipelines. Example failure cases from ToolFlow and ToolDial are shown in Appendix N and Appendix O respectively. ToolWeave dialogues also carry substantially more cross-tool dataflow, averaging 3.09 dataflow edges per dialogue against 0.41 for ToolFlow, where a dataflow edge links a tool-call argument to a value produced by an earlier tool response (Appendix H).

4.3 Hallucinations in Synthesized Dialogues

We randomly sampled 200 dialogues from ToolWeave, ToolDial, and ToolFlow to evaluate dataset quality. Table 3 reports different hallucination types and the percentage of dialogues exhibiting each error. A dialogue may contain multiple error types and thus appear in multiple rows, while the final row reports dialogues with no errors. The results highlight the benefits of detailed parameter planning in ToolWeave. The only hallucinations that persist in ToolWeave are value hallucinations. For

Table 3: Hallucination analysis across ToolWeave, ToolDial, and ToolFlow.

Hallucination Type	ToolWeave	ToolDial	ToolFlow
Tool name hallucination	0.0%	30.5%	5.0%
Parameter name hallucination	0.0%	3.5%	0.0%
Parameter value hallucination	20.0%	4.5%	46.5%
Assistant text output hallucination	0.0%	24.5%	75.5%
No hallucination (clean)	80.0%	46.5%	16.5%

Table 4: LLM evaluation of synthetic dialogues. Nat=Naturalness, Coh=Coherence, Hel=Helpfulness, Acc=Accuracy.

Dataset	Nat	Coh	Hel	Acc
ToolWeave	4.38	4.12	3.89	4.38
ToolFlow	3.85	2.94	3.21	3.27
ToolDial	3.38	3.42	3.00	3.82

example, when a user says *Charge the \$201.40 to my credit card*, the generated tool call may record *payment_amount: 20140*. Beyond turn-level hallucination, we also validate cross-tool dataflow: 94.2% of the 10,853 dataflow edges in ToolWeave dialogues carry values correctly traceable to an upstream tool response (Appendix H).

4.4 LLM-as-Judge Evaluation of Dialogues

Following prior work (Wang et al., 2025), we evaluate dialogue quality using an LLM-as-judge protocol. We use GPT-5-mini to rate 200 sampled dialogues per dataset on a 1–5 scale across four dimensions: *Naturalness* (human-likeness of the conversation), *Coherence* (logical flow and relevance), *Helpfulness* (usefulness of the assistant responses), and *Accuracy* (factual correctness). Table 4 reports the results. ToolWeave achieves the highest scores across all dimensions. In contrast, ToolFlow receives substantially lower coherence and helpfulness scores, consistent with the loop-heavy behavior observed in Table 2. ToolDial attains reasonable accuracy but lags in naturalness and coherence. Overall, these results confirm that ToolWeave produces dialogues that are both structurally complex and conversationally coherent. Appendix L.4 provides full protocol details and shows that these conclusions hold under three additional judge models.

5 Experiments

We evaluate ToolWeave along three questions:

1. *Performance Study*: How does ToolWeave compare with existing pipelines?

2. *Synthetic vs. Real APIs*: How do ToolWeave’s synthetic APIs compare with real APIs?
3. *Ablation Study*: How does each component of ToolWeave affect overall performance?

5.1 Experimental Setup

We synthesize ToolWeave data using GPT-5-mini, denoted as *ToolWeave (G)*, and GPT-0SS-120B, denoted as *ToolWeave (O)*. We use *ToolWeave-R* to denote variants that employ real-world RapidAPI schemas instead of our synthetic APIs. For each configuration, we generate 3.2K dialogues. We set $p_{clar} = 0.3$ and $p_{inject} = 0.7$ (Appendix L.3). We compare against two baselines: *ToolDial* (dialogues generated with GPT-4o) and *ToolFlow*². We focus on reproducible pipelines; several recent methods (Yin et al., 2025; Prabhakar et al., 2026; Liu et al., 2025) lack public code or do not specify the data mixtures used for fine-tuning, making faithful implementation and fair comparison difficult.

We evaluated three base models: *Llama-3.1-8B*, *Llama-3.1-70B*, and *Phi-4*, covering various model sizes and families. All models are fine-tuned using QLoRA via LLaMA-Factory. All hyperparameters used for fine-tuning are provided in Appendix J. We evaluate all models on a diverse suite of three multi-turn tool-calling benchmarks: *BFCL-V3* (Patil et al., 2025), *API Bank* (Li et al., 2023), and *CONFETTI* (Alkhoul et al., 2025). Benchmark details are provided in Appendix K.

Comparison scope: Our experiments isolate the effect of synthetic training data rather than claim leaderboard state of the art. We use controlled within-backbone comparisons, holding the base model and fine-tuning setup fixed while varying the training corpus among ToolDial, ToolFlow, and ToolWeave. Frontier proprietary systems score substantially higher on the continuously updated BFCL-V3 leaderboard (Patil et al., 2025), but are not directly comparable, as they differ in model scale, training data, and inference setup, and their rankings shift frequently.

5.2 Performance Study

BFCL-v3: Table 5 presents the results on BFCL v3 using the overall aggregated average accuracy (Avg. Acc). BFCL v3 evaluates not only the basic ability to perform tool calling (Base), but also tests whether a model can prompt the user when required information is missing (Miss Param.),

²We implemented ToolFlow using the prompts provided in the paper and synthesized the dialogues using GPT-5-mini.

Table 5: BFCL-V3 multi-turn performance.

Model	Avg. Acc.	Base	Miss Func.	Miss Param.	Long Ctx.
Llama-3.1-8B	9.25	11.00	8.00	8.50	9.50
+ ToolDial	1.75	0.50	3.50	2.50	0.50
+ ToolFlow	7.50	9.50	5.50	5.50	9.50
+ ToolWeave-R (G)	18.38	24.00	18.00	14.50	17.00
+ ToolWeave (O)	19.88	23.00	21.50	15.50	19.50
+ ToolWeave (G)	21.12	28.00	18.00	18.50	20.00
Llama-3.1-70B	12.50	17.00	13.00	10.50	9.50
+ ToolDial	3.75	4.50	3.00	4.00	3.50
+ ToolFlow	23.50	29.50	20.50	20.00	24.00
+ ToolWeave-R (G)	37.38	40.00	43.00	34.00	32.50
+ ToolWeave (O)	33.25	37.50	35.00	32.00	28.50
+ ToolWeave (G)	39.75	42.50	43.00	37.00	36.50
Phi-4	3.12	7.50	0.00	2.50	2.50
+ ToolDial	2.00	2.00	0.50	3.50	2.00
+ ToolFlow	10.12	11.00	10.00	11.50	8.00
+ ToolWeave-R (G)	26.25	33.50	33.50	20.50	17.50
+ ToolWeave (O)	24.50	24.00	28.50	31.00	14.50
+ ToolWeave (G)	29.50	36.00	30.50	31.50	20.00

Table 6: BFCL-V3 multi-turn performance of Llama-3.1-70B across two splits: multi-step and single-step dialogues.

	Multi-step Accuracy	Single-step Accuracy	Average Accuracy
Base	10.80	17.12	12.50
ToolFlow	19.75	26.32	23.50
ToolWeave	38.73	44.08	39.75

refrain from hallucinating when an appropriate tool is unavailable (Miss Func), and maintain accuracy in long, information-rich contexts (Long Ctx.). The BFCL dataset contains 800 dialogues in total, of which 648 include at least one multi-step interaction, while the remaining dialogues do not contain any multi-step interactions.

When LLMs are fine-tuned on dialogues generated by ToolDial, all three models perform worse than the base model. This is due to two key limitations: a cap of two tool calls per dialogue and a reliance on only 21 fixed interaction templates. As a result, only 4.47% of conversational turns in ToolDial are multi-step, with predefined flows that do not align well with the more realistic interaction patterns found in BFCL. ToolFlow performs better than ToolDial because it generates more realistic dialogues. However, its high-level planner places most of the burden on the dialogue synthesizer, resulting in only 6.77% of conversational turns being multi-step. In contrast, 44.87% of conversational turns in ToolWeave are multi-step, largely due to its

Table 7: Generalization across API-L1, API-L2, and CONFETTI.

Model	API-L1	API-L2	CONFETTI	Average
Llama-3.1-8B	62.91	59.46	20.55	47.64
+ ToolDial	59.15	52.70	23.12	44.99
+ ToolFlow	67.92	54.05	17.39	46.45
+ ToolWeave-R (G)	68.17	51.35	19.37	46.30
+ ToolWeave (O)	68.92	60.81	36.76	55.50
+ ToolWeave (G)	75.44	63.51	28.85	55.93
Llama-3.1-70B	54.89	59.46	33.00	49.12
+ ToolDial	71.93	40.54	11.46	41.31
+ ToolFlow	70.68	59.46	28.46	52.87
+ ToolWeave-R (G)	71.93	66.22	33.99	57.38
+ ToolWeave (O)	71.18	64.86	45.45	60.50
+ ToolWeave (G)	71.43	63.51	34.78	56.57
Phi-4	34.09	25.68	13.44	24.40
+ ToolDial	55.64	44.59	13.64	37.96
+ ToolFlow	49.12	56.76	15.61	40.50
+ ToolWeave-R (G)	75.19	62.16	32.41	56.59
+ ToolWeave (O)	71.68	62.16	33.60	55.81
+ ToolWeave (G)	76.69	64.86	31.42	57.66

fine-grained turn-level planner, thus outperforming ToolFlow by a large margin (at least 13 points). Table 6 shows the performance across two subsets of the data: multi-step and single-step. Across all three settings, multi-step accuracy consistently lags single-step by roughly 5-6 percentage points. This is expected as multi-step tasks compound errors at each step. ToolWeave achieves 3.6 times the multi-step accuracy and 2.6 times the single-step accuracy of the Base model.

Other Benchmarks: To assess the breadth of our approach, we evaluate ToolWeave across two additional benchmarks. Table 7 shows *ToolWeave* consistently improves performance across benchmarks, with average gains reaching 33.26 points (Phi-4). Notably, our 16.2 point improvement on CONFETTI demonstrates superior handling of complex conversational dynamics such as goal switching and chained function calls, where performance is typically limited across state of the art models. These results demonstrate that ToolWeave enhances multi-step reasoning and compositional tool use.

5.3 Synthetic vs. Real APIs

We further compare ToolWeave’s synthetic APIs against real APIs from RapidAPI. As shown in Table 5, Llama-3.1-70B achieves 39.75% accuracy when trained on fully synthetic ToolWeave data, compared with 37.38% when trained on RapidAPI-based data (ToolWeave-R). One possible explanation is that real-world APIs often lack explicit parameter dependency structure, which can intro-

duce noisy links in the tool graph. In contrast, ToolWeave’s synthetic tool graphs explicitly encode parameter relationships, producing a cleaner training signal for multi-step tool usage.

5.4 Ablation Study

Table 8: Component ablation on Llama-3.1-8B-Instruct (BFCL-V3).

Setting	Average Acc.
Full ToolWeave Pipeline	19.88
w/o Post-Processing	13.62
w/o Structured Sampler (Random Walk)	12.25
w/o Fine-Grained Planner (ToolFlow-style)	7.50

We perform ablation experiments on BFCL v3 using GPT-OSS-120B due to budget constraints. Table 8 reveals that each module in ToolWeave is critical for realistic multi-step tool-calling dialogue synthesis. Removing the fine-grained turn-level planner in favor of a high-level ToolFlow-style planner causes performance to collapse to 7.50%, confirming that models cannot synthesize complex trajectories without turn-level planning. Replacing our tool graph sampler with a random walk reduces accuracy to 12.25%. Removing post-processing leads to a 6.26% drop, validating the necessity of simulating real-world errors and recovery behaviors. Detailed breakdown of metrics for these ablations is provided in Appendix L.2.

6 Conclusion

We introduced ToolWeave, a modular framework for synthesizing realistic multi-turn tool-calling datasets through structured generation stages that enforce multi-step interactions and consistent parameter provenance. Across multiple LLM families and benchmarks, models fine-tuned on ToolWeave data significantly outperform existing pipelines. Our results reveal that structured synthesis enables open-weight models to produce higher-quality training data than frontier-model-driven pipelines. We further show that ToolWeave’s synthetic APIs can rival or surpass real-world APIs such as those from RapidAPI. Overall, our findings suggest that the structure of the synthesis pipeline, rather than the scale of the synthesis LLM, is the key driver of realistic multi-turn tool-calling data.

Limitations

While *ToolWeave* demonstrates significant gains in synthetic data quality, several limitations remain.

First, the framework relies on a *Tool Simulator* that produces schema-consistent but non-executable outputs; this enforces schema validity but not semantic validity (Appendix H), and cannot capture complex runtime side effects or hidden state changes present in real-world systems (e.g., database triggers or delayed failures); it also yields static trajectories for supervised fine-tuning rather than an interactive environment for online RL. Second, although ToolWeave’s scaffolding substantially reduces reliance on frontier reasoning, it still requires a synthesis LLM with basic instruction-following ability; models with extremely limited reasoning capacity may struggle to execute fine-grained plans, setting a lower bound on viable seed models. Third, ToolWeave currently focuses on text-based tool calling and does not yet support multimodal tools, such as agents interacting with GUIs or visual outputs, which are increasingly important for general agentic behavior. In addition, the multi-stage pipeline introduces higher computational overhead than one-shot generation approaches: generating 1,000 dialogues requires 30.2K LLM calls and 2.6 wall-clock hours, against 13.6K calls and 2.2 hours for ToolFlow (Appendix I). This is a one-time data generation cost that trades throughput for controllability and data quality. Finally, while ToolWeave enforces structural constraints and robustness patterns, it does not yet incorporate rich domain-specific policies or regulatory rules, and its workflow motifs and planning abstractions introduce inductive biases that may not cover all task structures. Addressing these limitations represents promising future work. Specifically, we plan to extend ToolWeave with fully executable synthetic APIs and broaden dialogue generation to capture richer agentic patterns such as ReAct (Yao et al., 2023). We also plan to incorporate *domain-specific policies* into synthesis, ensuring policy-aligned dialogues and robust instruction-following in rule-bound settings. Residual parameter-value errors are mostly formatting issues (e.g., dates, units), addressable via value normalization, schema validation, or trajectory skeletons (Sullivan et al., 2025).

References

- Tamer Alkhoul, Katerina Margatina, James Gung, Raphael Shu, Claudia Zaghi, Monica Sunkara, and Yi Zhang. 2025. [CONFETTI: Conversational function-calling evaluation through turn-level interactions](#). In *Proceedings of the 63rd Annual Meeting of*

the Association for Computational Linguistics (Volume 1: Long Papers), pages 7993–8006.

Kinjal Basu, Ibrahim Abdelaziz, Subhajit Chaudhury, Soham Dan, Maxwell Crouse, Asim Munawar, Veron Austel, Sadhana Kumaravel, Vinod Muthusamy, Pavan Kapanipathi, and Luis Lastras. 2024. [API-BLEND: A comprehensive corpora for training and benchmarking API LLMs](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12859–12870.

Kinjal Basu, Ibrahim Abdelaziz, Kiran Kate, Mayank Agarwal, Maxwell Crouse, Yara Rizk, Kelsey Bradford, Asim Munawar, Sadhana Kumaravel, Saurabh Goyal, Xin Wang, Luis A. Lastras, and Pavan Kapanipathi. 2025. [NESTFUL: A benchmark for evaluating LLMs on nested sequences of API calls](#). In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 33538–33547, Suzhou, China. Association for Computational Linguistics.

Jaime G. Carbonell and Jade Goldstein. 1998. [The use of MMR, diversity-based reranking for reordering documents and producing summaries](#). In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 335–336.

Mingyang Chen, Haoze Sun, Tianpeng Li, Fan Yang, Hao Liang, Keer Lu, Bin Cui, Wentao Zhang, Zenan Zhou, and Weipeng Chen. 2025a. [Facilitating multi-turn function calling for LLMs via compositional instruction tuning](#). In *The Thirteenth International Conference on Learning Representations*.

Yanfei Chen, Jinsung Yoon, Devendra Singh Sachan, Qingze Wang, Vincent Cohen-Addad, Mohammadhossein Bateni, Chen-Yu Lee, and Tomas Pfister. 2024. [Re-invoke: Tool invocation rewriting for zero-shot tool retrieval](#). In *Findings of the Association for Computational Linguistics: EMNLP 2024*, pages 4705–4726.

Yi-Chang Chen, Po-Chun Hsu, Chan-Jan Hsu, and Da-shan Shiu. 2025b. [Enhancing function-calling capabilities in LLMs: Strategies for prompt formats, data integration, and multilingual translation](#). In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 3: Industry Track)*, pages 99–111.

Bingguang Hao, Zengzhuang Xu, Maolin Wang, Yuntao Wen, Yicheng Chen, Cunyin Peng, Long Chen, Dong Wang, Xiangyu Zhao, Jinjie Gu, Chenyi Zhuang, and Ji Zhang. 2025. [Reasoning through exploration: A reinforcement learning framework for robust function calling](#). *Preprint*, arXiv:2508.05118.

Yue Huang, Jiawen Shi, Yuan Li, Chenrui Fan, Siyuan Wu, Qihui Zhang, Yixin Liu, Pan Zhou, Yao Wan, Neil Zhenqiang Gong, and Lichao Sun. 2024. [Meta-tool benchmark for large language models: Deciding](#)

- whether to use tools and which to use. In *The Twelfth International Conference on Learning Representations*.
- Vladimir I Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710.
- Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. 2023. **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.
- Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, Shuai Wang, Weinan Gan, Zhengying Liu, Yuanqing Yu, Zezhong Wang, Yuxian Wang, Wu Ning, Yutai Hou, Bin Wang, Chuhan Wu, Xinzhi Wang, Yong Liu, Yasheng Wang, and 8 others. 2025. **ToolACE: Winning the points of LLM function calling**. In *The Thirteenth International Conference on Learning Representations*.
- Zuxin Liu, Thai Hoang, Jianguo Zhang, Ming Zhu, Tian Lan, Shirley Kokane, Juntao Tan, Weiran Yao, Zhiwei Liu, Yihao Feng, Rithesh R. N., Liangwei Yang, Silvio Savarese, Juan Carlos Niebles, Huan Wang, Shelby Heinecke, and Caiming Xiong. 2024. **APIGen: Automated pipeline for generating verifiable and diverse function-calling datasets**. In *Advances in Neural Information Processing Systems Datasets and Benchmarks Track*.
- Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. 2024. **AgentBoard: An analytical evaluation board of multi-turn LLM agents**. In *Advances in Neural Information Processing Systems Datasets and Benchmarks Track*.
- Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. 2025. **The berkeley function calling leaderboard (BFCL): From tool use to agentic evaluation of large language models**. In *Forty-second International Conference on Machine Learning*.
- Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2024. **Gorilla: Large language model connected with massive APIs**. In *Advances in Neural Information Processing Systems*.
- Akshara Prabhakar, Zuxin Liu, Ming Zhu, Jianguo Zhang, Tulika Manoj Awalganekar, Shiyu Wang, Zhiwei Liu, Haolin Chen, Thai Quoc Hoang, Juan Carlos Niebles, Shelby Heinecke, Weiran Yao, Huan Wang, Silvio Savarese, and Caiming Xiong. 2026. **APIGen-MT: Agentic pipeline for multi-turn data generation via simulated agent-human interplay**. In *Advances in Neural Information Processing Systems Datasets and Benchmarks Track*.
- Shengqian Qin, Yakun Zhu, Linjie Mu, Shaoting Zhang, and Xiaofan Zhang. 2025. **Meta-tool: Unleash open-world function calling capabilities of general-purpose large language models**. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 30653–30677.
- Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, dahai li, Zhiyuan Liu, and Maosong Sun. 2024. **ToolLLM: Facilitating large language models to master 16000+ real-world APIs**. In *The Twelfth International Conference on Learning Representations*.
- RapidAPI. 2026. RapidAPI. <https://rapidapi.com/>. Accessed July 2025.
- Nils Reimers and Iryna Gurevych. 2019. **Sentence-BERT: Sentence embeddings using siamese bert-networks**. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992.
- Hayley Ross, Ameya Sunil Mahabaleshwarkar, and Yoshi Suhara. 2025. **When2Call: When (not) to call tools**. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3391–3409.
- S. Russell and P. Norvig. 2013. *Artificial Intelligence: Pearson New International Edition: A Modern Approach*. Pearson Education Limited.
- Timo Schick, Jane Dwivedi-Yu, Roberto Dessí, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. **Toolformer: Language models can teach themselves to use tools**. In *Advances in Neural Information Processing Systems*.
- Zhengliang Shi, Shen Gao, Lingyong Yan, Yue Feng, Xiuyi Chen, Zhumin Chen, Dawei Yin, Suzan Verberne, and Zhaochun Ren. 2025. **Tool learning in the wild: Empowering language models as automatic tool agents**. In *Proceedings of the ACM on Web Conference 2025*, pages 2222–2237.
- Jeonghoon Shim, Gyuhyeon Seo, Cheongsu Lim, and Yohan Jo. 2025. **ToolDial: Multi-turn dialogue generation method for tool-augmented language models**. In *The Thirteenth International Conference on Learning Representations*.
- Michael Sullivan, Mareike Hartmann, and Alexander Koller. 2025. **Procedural environment generation for tool-use agents**. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 18544–18562.
- Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. 2023. **ToolAlpaca: Generalized tool learning for language**

models with 3000 simulated cases. *arXiv preprint arXiv:2306.05301*.

Zezhong Wang, Xingshan Zeng, Weiwen Liu, Liangyou Li, Yasheng Wang, Lifeng Shang, Xin Jiang, Qun Liu, and Kam-Fai Wong. 2025. [ToolFlow: Boosting LLM tool-calling through natural and coherent dialogue synthesis](#). In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 4246–4263.

Mengsong Wu, Tong Zhu, Han Han, Chuanyuan Tan, Xiang Zhang, and Wenliang Chen. 2024. [Seal-Tools: Self-instruct tool learning dataset for agent tuning and detailed benchmark](#). In *CCF International Conference on Natural Language Processing and Chinese Computing*, pages 372–384. Springer.

Qiantong Xu, Fenglu Hong, Bo Li, Changran Hu, Zhengyu Chen, and Jian Zhang. 2023. [On the tool manipulation capability of open-source large language models](#). *Preprint*, arXiv:2305.16504.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2025. [\$\tau\$ -bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains](#). In *The Thirteenth International Conference on Learning Representations*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. [ReAct: Synergizing reasoning and acting in language models](#). In *The Eleventh International Conference on Learning Representations*.

Junjie Ye, Zhengyin Du, Xuesong Yao, Weijian Lin, Yufei Xu, Zehui Chen, Zaiyuan Wang, Sining Zhu, Zhiheng Xi, Siyu Yuan, Tao Gui, Qi Zhang, Xuanjing Huang, and Jiecao Chen. 2025. [ToolHop: A query-driven benchmark for evaluating large language models in multi-hop tool use](#). In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2995–3021.

Fan Yin, Zifeng Wang, I-Hung Hsu, Jun Yan, Ke Jiang, Yanfei Chen, Jindong Gu, Long Le, Kai-Wei Chang, Chen-Yu Lee, Hamid Palangi, and Tomas Pfister. 2025. [MAGNET: Multi-turn tool-use data synthesis and distillation via graph translation](#). In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 32600–32616.

Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. 2023. [ToolQA: A dataset for LLM question answering with external tools](#). In *Advances in Neural Information Processing Systems Datasets and Benchmarks Track*.

A Use of AI Assistants

In preparing this paper, we made limited use of Large Language Models (LLMs) as writing assistance tools. Specifically, we used Grammarly³ for spell checking and grammar correction to improve clarity and readability. In line with the conference policy, we confirm that LLMs were not used for generating research ideas or making any substantive contributions to the scientific content of this work.

B Comparison of ToolWeave with other SDG pipelines

Table 9 provides the comparison of the features around licenses of the model used, the code, the generated data, and the fine-grained control it offers.

C Tool Graph Synthesizer Details

This section provides additional implementation details of the Tool Graph Synthesizer, including the prompts used for generating APIs, schema validation procedures, and graph construction strategies.

C.1 Prompts Used

All prompts used in the tool graph synthesis stage are available in the prompts directory of the code repository.⁴

C.2 Domains for Tool Graph Synthesis

The 20 domains used to generate the synthetic API pool are:

Agriculture	Logistics
Customer Support	Manufacturing
Cybersecurity	Marketing
E-commerce	Online Banking
Education	Real Estate
Energy	Retail
Film Industry	Supply Chain
Human Resources	Telecommunications
Insurance	Tourism
Legal Services	Transportation

C.3 Sample Tool Graph

Figure 4 illustrates a representative tool graph for the E-commerce domain. The nodes represent discrete tools (APIs), connected by directed edges that signify data dependencies; specifically, a parameter from the output schema of a predecessor node maps to an input parameter of the successor

³<https://app.grammarly.com/>

⁴<https://github.com/IBM/ToolWeave>

Algorithm 1: Tool Graph Synthesis

Input: Domain descriptor D , synthesis plan P , LLM L , embedding model E
Output: API pool $\mathcal{A}$, Tool Graph $\mathcal{G}$
Initialize $\mathcal{A} \leftarrow \emptyset$, $\mathcal{G} \leftarrow \emptyset$;
 $C \leftarrow \text{CONSTRUCTDOMAINCONTEXT}(D)$; // See Appendix C.6.1
foreach step $s \in P$ **do**
 Generate n_s candidate APIs with L using domain context C and prompt from step s ;
 Parse and validate candidates syntactically;
 Deduplicate candidates against $\mathcal{A}$ using embeddings from E and structural checks;
 Refine and semantically validate schemas (defaults, enums, nested objects, etc.);
 Add accepted APIs to $\mathcal{A}$;
Construct Tool Graph $\mathcal{G}_{tool}$ by validating data-flows between APIs in $\mathcal{A}$;
return $\mathcal{A}$, $\mathcal{G}_{tool}$

(e.g., zone_policy from get_geo_rules acts as an input for set_mode).

The graph topology highlights three distinct execution patterns, corresponding to the Tool-Graph traversal strategies discussed in Section 3.2:

- **Linear Chains:** Standard sequential dependencies connected by any type of edge, such as $\text{get_order} \rightarrow \text{set_mode} \rightarrow \text{ship_local} \rightarrow \text{save_track}$.
- **Fan-in-Fan-out (Dashed):** Illustrated by the sequence $\text{get_order} \rightarrow \{\text{check_stock}, \text{calc_risk}\} \rightarrow \text{sync_status}$. Here, the initial tool provides inputs to a parallel stage of tools that are independent of one another, the outputs of which collectively feed into a common successor.
- **Conditional Branching (Green):** Represented by the set_mode node, where the downstream execution path (ship_local vs. ship_intl) is determined dynamically based on the output value of the decision node.

C.4 Detailed Algorithms for Tool Graph Synthesizer

This section expands upon the Tool Graph Synthesizer process from Section 3.1. While the description in the main paper presents the high-level methodology, here we provide the main algorithm, the synthesis plan, and detailed sub-algorithms for key steps like domain context construction and API refinement.

Table 9: Comparison of multi-turn synthetic data pipelines across different dimensions. A $\checkmark^*$ under *License Friendly Synthesis Model* indicates that the pipeline has been demonstrated to produce high-quality data with an open-weight synthesis model. We report ToolWeave with both GPT-OSS-120B (open weight) and GPT-5-mini, whereas prior pipelines report results only with proprietary frontier models.

Framework	Synthesize APIs	Code Released	Data Released, License Friendly	License Friendly Synthesis Model	Fine-Grained Plan
Tooldial (Shim et al.)	$\times$	$\checkmark$	$\checkmark, \times$	$\times$	$\times$
ToolAce (Liu et al.)	$\checkmark$	$\times$	$\checkmark, \times$	$\times$	$\times$
APIgen-MT (Prabhakar et al.)	$\times$	$\times$	$\checkmark, \times$	$\times$	$\times$
ToolFlow (Wang et al.)	$\times$	$\times$	$\times, \times$	$\times$	$\times$
Button (Chen et al.)	$\checkmark$	$\times$	$\checkmark, \times$	$\times$	$\times$
ToolLLM (Qin et al.)	$\times$	$\checkmark$	$\checkmark, \times$	$\times$	$\times$
ToolWeave (Ours)	$\checkmark$	$\checkmark$	$\checkmark, \checkmark$	$\checkmark^*$	$\checkmark$

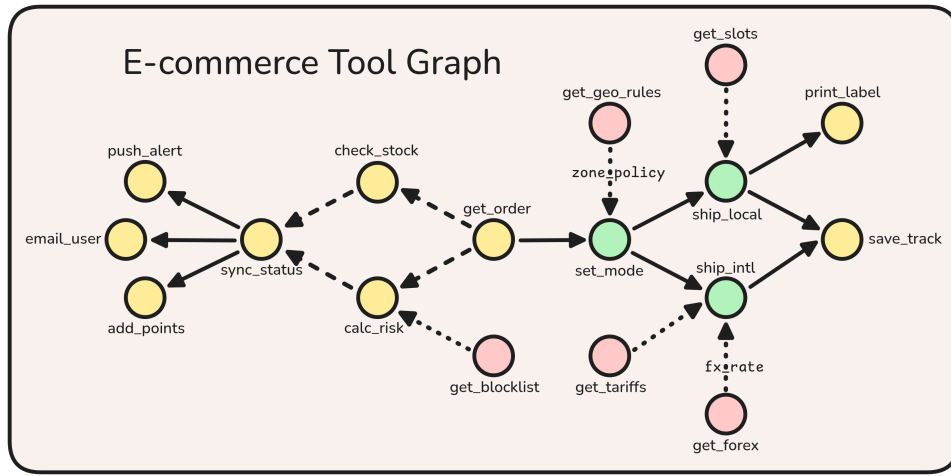


Figure 4: E-commerce tool graph demonstrating linear, fan-in-fan-out (dashed), and conditional (green) dependency patterns. Red nodes indicate auxiliary tools providing specific configuration parameters, such as foreign exchange rates or zone policies, to the primary workflow.

C.5 Synthesis Plan

Our synthesis plan acts as a curriculum for the API-generating LLM. It specifies a sequence of generation steps, each with a unique prompt template and a target number of APIs. This design is extensible and allows us to control the breadth and depth of the final API pool. Our implementation consists of five main stages:

1. **Seed Generation:** This initial stage generates a small batch of fundamental, “entry-point” APIs for the domain (e.g., `search_product` in e-commerce).
2. **Entity Expansion:** This stage uses the entities extracted from Wikidata (see Appendix C.6.1) to generate new APIs that specifically cover domain-relevant entities, ensuring breadth.
3. **Schema Enrichment:** This stage focuses on depth. It takes existing, simple APIs and iteratively

increases their structural complexity by adding nested objects, enums, default values, and required flags to mimic real-world enterprise-grade schemas.

4. **Connection Discovery:** It generates new APIs that plausibly connect existing APIs, creating data-flow paths (e.g., a `get_product_details` API that takes a `product_id` from `search_product`).
5. **Pattern Expansion:** This final stage diversifies the API pool by generating parallel variations of existing APIs (e.g., different ways to search, like `search_by_name`, `search_by_category`, etc.).

For reproducibility, we include the synthesis plan used in our experiments below. Prompt paths have been abstracted for clarity.

```

{
  "steps": [
    {"name": "Seed Generation", "num_to_generate": 8},
    {"name": "Entity Expansion", "num_to_generate": 8},
    {"name": "Schema Enrichment", "num_to_generate": 5},
    {"name": "Connection Discovery", "num_to_generate": 8},
    {"name": "Pattern Expansion", "num_to_generate": 5}
  ]
}

```

Algorithm 2: CreateDomainContext

Input: Domain name string d
Output: Domain context C
 Resolve d to canonical Wikidata entity QID ;
 Retrieve Wikipedia summary T_{wiki} for d ;
 Extract structured facts $F_{wikidata}$ (classes, subclasses, properties) from QID ;
 Assemble $C \leftarrow (T_{wiki}, F_{wikidata})$;
return C

C.6 Detailed Sub-Algorithms

The main generation algorithm (Algorithm 1) relies on several key sub-routines, which are detailed below.

C.6.1 Domain Context Construction

To provide domain grounding for API synthesis, we construct a domain context using external knowledge sources (Algorithm 2). Specifically, we retrieve a Wikipedia summary and structured metadata from Wikidata, including entity descriptions, subclasses, and related properties. These signals are consolidated into a compact prompt representation used during generation. The resulting context constrains the synthesis process to domain-relevant entities, operations, and relationships.

C.6.2 API Refinement

Generated APIs are post-processed to improve schema consistency and usability (Algorithm 3). This stage removes redundant or echo parameters, resolves required fields, and refines parameter properties such as enums, default values, and formats. Duplicate APIs are filtered using lexical, structural, and semantic similarity checks. These steps produce a consistent and non-redundant API pool for subsequent workflow construction.

C.6.3 Connection Graph Construction

Candidate connections are identified when an output parameter of one API and an input parameter of another share the same base name; this is effective in our setting because the synthesis prompts instruct

Algorithm 3: RefineAPI

Input: Candidate API a , current API set $\mathcal{A}$
Output: Refined API a' or $\emptyset$
 Check if a is duplicate of any in $\mathcal{A}$ (lexical, structural, semantic);
if duplicate then
 return $\emptyset$
 Remove echo parameters (outputs attributes that are same as inputs);
 Enrich schema: infer enums, default values, required fields;
 Normalize parameter names and types;
 Paraphrase parameter names and descriptions for stylistic variance;
return refined API a'

Algorithm 4: ConstructToolGraph

Input: API set $\mathcal{A}$
Output: Connection graph $\mathcal{G}$
 Initialize graph $\mathcal{G} = (V = \mathcal{A}, E = \emptyset)$;
foreach ordered pair $(a_i, a_j) \in \mathcal{A} \times \mathcal{A}$ **do**
 Identify candidate parameter matches (p_{out}, p_{in})
 ; // Shared base name: p_{out} in a_i 's
 outputs, p_{in} in a_j 's inputs
 Validate (p_{out}, p_{in}) by description similarity or LLM judgment;
 if valid then
 Add edge $(a_i.p_{out} \rightarrow a_j.p_{in})$ to E ;
return $\mathcal{G}$

the model to connect new APIs to previously generated ones through identically named parameters (Algorithm 4). Each candidate is then validated by comparing the two parameter descriptions using sentence-embedding similarity, or optionally by LLM judgment, before the edge is added. A subsequent paraphrasing stage renames a fraction of parameters, prioritising those involved in connections, and updates the graph accordingly. Connected parameters in the released APIs therefore typically carry different names (e.g., `feedback_id` $\rightarrow$ `feedback_record_id`), which prevents models from resolving dependencies by surface string matching alone.

C.7 API Quality Metric Definitions

This section provides the formal definitions for the metrics used to evaluate the quality of the synthetic APIs in Section 4.1. In all definitions, $A = \{a_1, a_2, \dots, a_n\}$ represents the set of n APIs in a given domain.

1. Interconnectivity (IC)

The average number of input parameters per API that can be filled by an output from another API in the graph. This measures the data-flow potential.

$$IC = \frac{\sum_{i=1}^n |I_{a_i} \cap O|}{n}$$

Where:

- $O = \bigcup_{i=1}^n O_{a_i}$: The set of all unique output parameters from all APIs.
- I_{a_i} : The set of input parameters for API a_i .
- $|I_{a_i} \cap O|$: The count of input parameters for a_i that match any output from the entire API set A .

2. Complex API Use (CAU)

The proportion of APIs that use at least one complex parameter type (i.e., “object” or “array”), measuring schema depth.

$$CAU = \frac{\sum_{i=1}^n C(a_i)}{n}$$

Where the indicator function $C(a_i)$ is defined as:

$$C(a_i) = \begin{cases} 1, & \text{if } a_i \text{ has an object or array,} \\ & \text{parameter} \\ 0, & \text{otherwise} \end{cases}$$

3. Required Parameter Ratio (RPR)

The average proportion of an API’s input parameters that are marked as “required”, measuring schema strictness.

$$RPR = \frac{1}{n} \sum_{i=1}^n \frac{|I_{\text{req}}(a_i)|}{|I_{\text{total}}(a_i)|}$$

Where $I_{\text{req}}(a_i)$ is the set of required input parameters for API a_i , and $I_{\text{total}}(a_i)$ is the set of all input parameters for a_i .

4. Longest Chain Length

The length of the longest simple path (a path with no repeated nodes) in the *Tool Graph*. This metric quantifies the maximum number of sequential, multi-step operations possible in a single dialogue.

C.8 Human Evaluation of Tool Graph Edges

To assess whether tool graph edges encode genuine task dependencies rather than surface schema compatibility, we compare the ToolWeave tool graph against the ToolDial tool graph (Shim et al., 2025), which is constructed from RapidAPI schemas using parameter-level semantic similarity.

Protocol One of the authors annotated 100 edges, 50 sampled from each graph, presented in randomized order with the source system hidden. For each edge, the annotator judged whether it forms a semantically coherent workflow, that is, whether the source tool genuinely produces output that the target tool requires, such that the downstream call cannot meaningfully proceed without the upstream one.

Results 74% of ToolWeave edges were judged coherent, against 12% of ToolDial edges. ToolWeave’s Connection Discovery stage (Section 3.1) generates linking parameters between tools by design, and each candidate edge is validated by an LLM judge before being added to the graph (Algorithm 4). The resulting edges encode real business-process constraints:

- `initiate_transfer` → `approve_transfer`, via `transfer_id`
- `create_retainer_agreement` → `generate_onboarding_checklist`, via `retainer_id`

ToolDial instead infers edges from parameter-level similarity, which connects tools that merely share an input. For example, it links

- `seasonal_goal_analysis_scoring_first`
 - `seasonal_goal_analysis_conceding_first`
- which both take the same `matchId`, but neither produces information the other requires, so there is no reason to call one before the other in a workflow. The difference is reflected downstream: ToolDial yields 4.47% multi-step turns against ToolWeave’s 44.87% (Table 2).

D Goal Generation Implementation Details

This section provides the low-level implementation details for the *Goal Generation* stage (Section 3.2), including the algorithms and heuristics used to extract, score, and synthesize goals.

D.1 Prompts Used

All prompts used in the tool graph sampler and goal generation stage are available in the prompts directory of the [code repository](#).

D.2 Pattern Extraction Algorithms

The main paper outlines three pattern classes. Here is how they are discovered:

1. **Linear Paths:** These are found using a bounded *beam search* (Russell and Norvig, 2013). The search explores paths from a start node, expanding a “beam” of the top- k candidates at each depth.
2. **Fan-in / Fan-out:** These patterns are found by analyzing node connectivity. For a given `start_node`, we find its successors. We then find the “common children” of combinations of these successors using set intersection. This allows us to identify points where parallel data flows (fan-out) later merge (fan-in).
3. **Conditional Branches:** These are identified by scanning an API’s output schema. We scan for output fields that can act as a logical predicate, specifically those typed as `boolean`, `enum`, or other simple types, while filtering out non-conditional fields like IDs.

D.3 Path Scoring and Ranking

To rank the paths found by our algorithms, we use a combination of a hybrid score and a diversification algorithm.

Hybrid Heuristic Score: The `final_score` for any given path is a weighted sum of three components, and is calculated as:

$$S_{\text{final}} = w_l \cdot (S_{\text{coherence}} + S_{\text{relevance}}) + w_d \cdot S_{\text{dataflow}} + w_b \cdot S_{\text{length}}$$

Where:

- $S_{\text{coherence}}$ and $S_{\text{relevance}}$ are *LLM-based ratings* (from -2 to $+2$) where the LLM judges the quality of a synthesized goal for the provided tool path.
- S_{dataflow} is the *semantic dataflow score*, calculated by checking cosine similarity between embeddings of output and input parameters of sequential tools.
- S_{length} is a *length bonus* to reward longer, more complex paths.
- w_l , w_d , and w_b are the respective weights for LLM rating, dataflow, and length, set as hyperpa-

rameters (we use $w_l = 0.5$, $w_d = 0.8$, $w_b = 0.3$ in our experiments).

Ranking and Embedding Model: To ensure a diverse set of goals, we rank the top-scoring linear paths using *Maximal Marginal Relevance (MMR)* (Carbonell and Goldstein, 1998). This balances the `final_score` (relevance) with the cosine dissimilarity from already-selected goals (diversity). In our experiments, we use a value of 0.7 for λ . All embeddings for MMR and semantic dataflow are computed using the `all-MiniLM-L6-v2` model from the *SentenceTransformers* (Reimers and Gurevych, 2019) library.

D.4 Goal Generation Algorithm

The goal generation algorithm detailed in Algorithm 5 converts sampled tool paths into natural-language task descriptions. Given a sequence of tools, the algorithm constructs a goal that reflects the functional intent of the workflow while preserving dependencies between tool inputs and outputs. Candidate goals are generated and filtered to ensure semantic coherence with the underlying tool sequence. This procedure associates each executable tool path with a clear user-oriented objective.

D.5 Example Goal Synthesis Prompt

The final step, ‘Goal synthesis via LLM’, uses different prompts for each pattern type. The prompt provides the LLM with the tool sequence and their schemas, and asks it to generate a natural-language goal.

E Dialogue Planner Implementation Details

This section provides the low-level implementation details for the *Dialogue Planner* stage (Section 3.3), including the algorithm and the logic used to partition tool paths, generate subgoals, plan parameters, and inject interaction patterns.

E.1 Prompts Used

All prompts used in this planner stage are available in the prompts directory of the [code repository](#).

E.2 Dialogue Planner Algorithm

Algorithm 6 summarizes the high-level process, highlighting subgoal generation, speaker sequencing, parameter planning, and clarification insertion.

Algorithm 5: Domain-level Goal Generation

Input: Tool graph for domain $\mathcal{G}_{tool}$, beam width B , max depth D_{max} , MMR weight λ , score threshold τ , LLM L , embedding model E

Output: Goal set $\mathcal{G}_{goals}$ (JSON objects: {tool_path, pattern_type, goal_text, metadata})

Initialize $\mathcal{G}_{goals} \leftarrow \emptyset, S \leftarrow \emptyset$, where S is the set of selected tool paths;

$S_{linear} \leftarrow \text{BEAMSEARCH}(\mathcal{G}_{tool}, B, D_{max}, \tau, \lambda, L, E)$;

// at each depth: generate a goal per candidate path, score it, discard below τ , MMR-prune the beam

$S_{fan} \leftarrow \text{FINDFANINFANOUT}(\mathcal{G}_{tool})$;

$S_{cond} \leftarrow \text{FINDCONDITIONAL}(\mathcal{G}_{tool})$;

foreach $p \in S_{fan} \cup S_{cond}$ **do**

$prompt \leftarrow \text{BUILDGOALPROMPT}(p, \text{PATTERN_TYPE}(p))$;

$goal_text \leftarrow L.GENERATE(prompt)$;

$SCORE(goal_text, p)$;

$S \leftarrow S_{linear} \cup S_{fan} \cup S_{cond}$;

foreach $p \in S$ **do**

$obj \leftarrow \text{BUILDJSON}(p, \text{PATTERN_TYPE}(p), \text{GOAL_TEXT}(p), \text{METADATA}(p))$;

$\mathcal{G}_{goals} \leftarrow \mathcal{G}_{goals} \cup \{obj\}$;

return $\mathcal{G}_{goals}$

E.3 Tool Path Partitioning

The PARTITIONTOOLPATH function uses different strategies based on the pattern type:

- **Linear:** We use an LLM-based approach. The prompt provides the full tool path and their schemas, and asks the LLM to split them into coherent segments, based on logical groupings and dataflow, while also ensuring meaningful multi-step tool sequences.
- **Fan-in / Fan-out:** We use a deterministic graph-based approach. We identify the branching point (fan-out) and the merging point (fan-in) in the path, and create partitions accordingly. The intermediate tools between these points are grouped into either segment, and all possible valid partitions are generated to allow for diverse dialogue structures.
- **Conditional:** Based on the value of the decision variable (from goal metadata M), the second tool is chosen from the available branches. The first tool and this chosen branch form a single partition to ensure the conditional logic is preserved in the dialogue, while also generating high-quality multi-step dialogues.

E.4 Subgoal Synthesis

The SYNTHESIZESUBGOAL function uses an LLM prompt that provides the tool schemas in the partition and the overall goal text G . The LLM is instructed to generate a concise, user-friendly subgoal that accurately reflects the purpose of the tools in the partition, while ensuring it is distinct from other subgoals in the dialogue.

E.5 Parameter Planning and Clarification Insertion

The parameter planning and clarification insertion process involves several steps to ensure that the dialogue captures realistic user-assistant interactions.

E.5.1 Parameter Preparation

The EXTRACTTOOLPARAMS function retrieves all input parameters for the tools in the current partition. The UPSTREAMTOOLOUTPUTS function identifies any parameters that can be automatically filled from outputs of tools in previous partitions, and these are removed from the required parameters.

Algorithm 6: Dialogue Planning

Input: Tool path P , pattern type $T \in \{\text{linear}, \text{fan}, \text{conditional}\}$, overall goal text G , relevant tool schemas Σ , goal metadata M , clarification probability p_{clar} , LLM for partitioning and subgoal synthesis L

Output: Ordered plan $\mathcal{D}$ (step-level JSON objects with keys: `step_idx`, `role`, `subgoal`, `tools`, `params`, `metadata`)

Initialize partition list $\mathcal{P}^* \leftarrow \text{PARTITIONTOOLPATH}(P, T, \Sigma, L)$, subgoal text list $\mathcal{S} \leftarrow []$;

foreach partition $p \in \mathcal{P}^*$ **do**

$\mathcal{S}.\text{APPEND}(\text{SYNTHESIZESUBGOAL}(p, G, L))$;

$\mathcal{D} \leftarrow []$;

for $i \leftarrow 1$ **to** $|\mathcal{P}^*|$ **do**

$p \leftarrow \mathcal{P}^*[i]$, $s \leftarrow \mathcal{S}[i]$;

$\text{Params} \leftarrow \text{EXTRACTTOOLPARAMS}(p, \Sigma)$;

$\text{Params} \leftarrow \text{Params} \setminus \text{UPSTREAMTOOLOUTPUTS}(\mathcal{P}^*, i - 1, \Sigma)$;

$(\text{ReqParams}, \text{ClarParams}) \leftarrow \text{SPLITPARAMS}(\text{Params}, p_{clar})$;

$\mathcal{D}.\text{APPEND}(\text{CREATEUSERTURN}(i, s, p, \text{ReqParams}))$;

if $\text{ClarParams} \neq \emptyset$ **then**

$\mathcal{D}.\text{APPEND}(\text{CREATEASSISTANTCLARIFICATIONTURN}(i, \text{ClarParams}))$;

$\mathcal{D}.\text{APPEND}(\text{CREATEUSERCLARIFICATIONTURN}(i, \text{ClarParams}))$;

foreach tool $t \in p$ **do**

$\text{ToolParams} \leftarrow \text{EXTRACTTOOLPARAMS}([t], \Sigma)$;

$\mathcal{D}.\text{APPEND}(\text{CREATEASSISTANTTOOLCALLTURN}(i, t, \text{ToolParams}, M))$; // Goal metadata M attaches decision variables for conditionals

$\mathcal{D}.\text{APPEND}(\text{CREATEASSISTANTSUMMARYTURN}(i, p))$;

return $\mathcal{D}$

E.5.2 Clarification Insertion

The `SPLITPARAMS` function randomly selects a subset of the remaining parameters to be clarified, based on the clarification probability p_{clar} . This introduces variability and simulates real-world scenarios where users may not provide all necessary information upfront.

E.5.3 Speaker Sequencing and Turn Creation

The turn creation functions add well-formatted JSON objects to the dialogue plan, specifying the role and the associated content (subgoal, tool calls, parameter requests, clarifications, summaries). Following is the structure of these turns:

- `CreateUserTurn`: User turn with subgoal and parameters to provide upfront.
- `CreateAssistantClarificationTurn`: Assistant turn asking for clarification on missing pa-

rameters required for the tool calls in the current partition.

- `CreateUserClarificationTurn`: User turn providing values for the requested parameters.
- `CreateAssistantToolCallTurn`: Assistant turn invoking a tool with the provided parameters and source hints for each parameter, either from the user or from upstream tool outputs.
- `CreateAssistantSummaryTurn`: Assistant turn summarizing the results of the tools in the current partition.

F Dialogue Synthesizer Implementation Details

This section provides the low-level implementation details for the *Dialogue Synthesizer* stage (Section 3.4), including the overall algorithm and the logic used to build context for each role across

agents.

F.1 Prompts Used

The prompts associated with each agent role in this stage, including user message paraphrasing during post-processing and those for dialogue synthesis and refinement, are available in the prompts directory of the [code repository](#).

F.2 Dialogue Synthesis Algorithm

Algorithm 7 summarizes the high-level process, highlighting context construction, dynamic role-based agent selection, response generation, and memory updates.

G Dialogue Post-Processing

G.1 Dialogue Error Injection

To rigorously evaluate and improve the robustness of models trained on ToolWeave data, we implemented a systematic error injection pipeline. This pipeline deterministically introduces schema violations, logical ordering errors, and tool confusion scenarios to force the model to learn error recovery in simulated real-world settings. Algorithm 8 details the high-level injection logic, while Algorithm 9 illustrates the logic required to synthesize cascading failure scenarios.

Additional Error Injection Modules: Beyond the cascading failures detailed in Algorithm 9, our pipeline includes:

- **Out-of-Order Execution:** We simulate scenarios where the model attempts to execute a tool call before its dependencies are met. Specifically, we identify dependent tool pairs and inject a premature call to the downstream tool (missing the dependent parameter) prior to the prerequisite tool. The model receives a missing dependency error and must recover by executing the sequence in the correct order.
- **Semantic Confusion (Wrong Tool):** We utilize a hybrid similarity matcher (SentenceBERT (Reimers and Gurevych, 2019) + Levenshtein distance (Levenshtein, 1966)) to identify semantically similar but incorrect tools (e.g., `search_ticket` vs. `get_ticket_details`). We inject a turn where the model mistakenly calls the confusable tool, receives an unhelpful response, and must self-correct.
- **Schema Violations:** We deterministically mutate valid tool calls to violate JSON schema constraints. This includes removing required param-

eters (MISSING_PARAM), violating type constraints (e.g., passing an integer for a string field), or using invalid enum values. The model is trained to read the resulting standardized error message and retry with the correct schema.

- **Missing Function Recovery:** We simulate scenarios where a necessary tool is initially absent from the provided context. The algorithm identifies a scheduled tool call in the plan and temporarily “hides” it. We inject an Assistant turn refusing the request due to the missing tool, followed by a User turn that explicitly provides the missing tool’s schema JSON. The model is trained to recognize the new tool definition in the context window and immediately proceed with the correct tool call.

H Dataflow Validation

A dataflow edge exists when a tool-call argument should carry a value produced by an earlier tool response. For example, if `create_booking` returns `booking_id = "bkg_123"` and a later call passes `reservation_reference = "bkg_123"`, the two calls are connected by one dataflow edge.

Detection For ToolWeave we identify dataflow edges from the parameter-level provenance recorded in the plan (Section 3.3). ToolFlow records no such provenance, so we use Claude Opus 4.7 to identify its edges instead. The two pipelines are therefore measured with different detection procedures, which we note as a caveat on the comparison.

Coverage ToolWeave averages 3.09 dataflow edges per dialogue against 0.41 for ToolFlow, at 3.22 and 2.20 tool calls per dialogue respectively (Table 2), indicating a substantially richer multi-step dependency structure rather than merely more tool calls.

Accuracy Across 10,853 ToolWeave dataflow edges we find 633 violations, giving 94.2% dataflow accuracy. Table 10 categorizes these violations into interpretable error types, which enables automatic filtering of inconsistent samples.

Example violation The tool `search_tickets` returns three tickets with identifiers TKT-2025-4491, TKT-2025-4510, and TKT-2025-4463. The following call to `assign_ticket` passes all three as a single concatenated string, `support_ticket_id = "TKT-2025-4491, TKT-2025-4510, TKT-2025-4463"`, where the

Algorithm 7: Plan-Driven Dialogue Synthesis

Input: Ordered plan $\mathcal{D}$, tool schemas Σ , language models for agents L

Output: Dialogue transcript $\mathcal{T}$

Initialize memory $\mathcal{M} \leftarrow \emptyset$, transcript $\mathcal{T} \leftarrow []$;

foreach *step* $d \in \mathcal{D}$ **do**

```
     $ctx \leftarrow \text{BUILDCONTEXT}(\mathcal{M}, d, \Sigma)$  ;           // Get memory and plan step
     $role \leftarrow d.role$ ;
     $agent\_prompt \leftarrow \text{SELECTPROMPT}(role)$  ;       // Prompt with agent's role
     $response \leftarrow L.GENERATE(agent\_prompt, ctx)$  ;   // Utterance/tool call
     $\mathcal{T}.APPEND(response)$ ;
     $\mathcal{M} \leftarrow \text{UPDATEMEMORY}(\mathcal{M}, response, d)$  ;      // Update memory with output
```

return $\mathcal{T}$

Algorithm 8: Dialogue Error Injection

Input: Dialogue set $\mathcal{D}$, Tool Schemas $\mathcal{S}$, Injection Probability p_{inject} , Similarity Matcher M

Output: Augmented Dialogue set $\mathcal{D}_{aug}$

Initialize $\mathcal{D}_{aug} \leftarrow \emptyset$;

foreach *dialogue* $d \in \mathcal{D}$ **do**

```
    Add original  $d$  to  $\mathcal{D}_{aug}$ ;
    if  $\text{Random}(0, 1) < p_{inject}$  then
         $d_{error} \leftarrow \text{DeepCopy}(d)$ ;
         $injected \leftarrow \text{False}$ ;
        // Attempt Complex Logical Errors (30% chance)
        if  $\text{Random}(0, 1) < 0.3$  then
            if  $\text{INJECTCASCADINGFAILURE}(d_{error}, \mathcal{S})$  is Success then
                 $injected \leftarrow \text{True}$ ;
            else if  $\text{INJECTOUTOFORDERERROR}(d_{error}, \mathcal{S})$  is Success then
                 $injected \leftarrow \text{True}$ ;
            else if  $\text{INJECTWRONGTOOLERROR}(d_{error}, \mathcal{S}, M)$  is Success then
                 $injected \leftarrow \text{True}$ ;
        // Attempt Simple Parameter Errors (70% chance or if complex failed)
        if not injected then
             $\text{INJECTSCHEMAERROR}(d_{error}, \mathcal{S})$ ;
        if error injected successfully then
            Mark  $d_{error}$  as modified;
            Add  $d_{error}$  to  $\mathcal{D}_{aug}$ ;
```

return $\mathcal{D}_{aug}$

tool expects one identifier. Each individual value appears upstream, but the combined value corresponds to no single element. The error arises during dialogue generation; the simulator accepts the input because it is schema-valid (a string) and does not enforce semantic validity.

I Generation Cost

Table 11 reports the cost of generating 1,000 dialogues with ToolWeave and, under identical infrastructure, with ToolFlow. Both pipelines run as CPU jobs issuing requests to a vLLM inference server, using thread-pool parallelism where applicable.

ToolWeave requires 30.2K LLM calls and 2.6 wall-clock hours to generate 1,000 dialogues, against 13.6K calls and 2.2 hours for ToolFlow. The

Algorithm 9: Inject Cascading Failure

Input: Dialogue d , Tool Schemas $\mathcal{S}$
Output: Modified Dialogue d' , Success boolean
Identify long multi-step sequences T_{seq} ($\text{len} \geq 3$) in d ;
if T_{seq} is empty **then**
 return d , *False*
Select random sequence $S \in T_{seq}$;
// 1. Simulate Reverse-Order Execution
Initialize $\text{failure_turns} \leftarrow \emptyset$;
for $i \leftarrow \text{length}(S)-1$ **down to** 1 **do**
 $\text{step}_{curr} \leftarrow S[i]$, $\text{step}_{prev} \leftarrow S[i-1]$;
 Identify param p_{dep} in step_{curr} dependent on step_{prev} ;
 Create turn t_{call} : Call step_{curr} **missing** p_{dep} ;
 Create turn t_{err} : Error "Prerequisite step not completed/Missing input";
 Append (t_{call}, t_{err}) to failure_turns ;
// 2. Inject Full Recovery
Construct new history: $\text{history}_{original} + \text{failure_turns} + \mathcal{S}_{original}$;
Update $d'.\text{conversations}$ and re-index $d'.\text{plan}$ with error steps;
return d' , *True*

Table 10: Categorization of the 633 dataflow violations found among 10,853 dataflow edges in ToolWeave dialogues.

Category	Count	%	Description
Paraphrased	239	37.8	Downstream rewrites or summarizes the upstream value instead of copying it
Fabricated	165	26.1	Downstream invents a plausible value that appears nowhere upstream
Wrong Selection	130	20.5	Upstream returns an array; downstream picks a value not present in any element
Missing Path	99	15.6	The upstream output lacks the declared field entirely (e.g. an empty array)
Total	633	100.0	

additional cost reflects ToolWeave’s extra pipeline stages (API synthesis, goal generation, and dialogue refinement), together with its longer multi-step dialogues. Table 11 gives the per-stage breakdown. Note that this is a one-time data generation cost, not a recurring inference cost.

J Fine-tuning Hyperparameters

We fine-tune all models on 8 NVIDIA A100 SXM4 80GB GPUs using QLoRA via the LLaMA-Factory framework⁵. The primary hyperparameters used for each model are detailed in Table 12. Common settings across all models include a learning_rate of 5.0×10^{-5} , bf16 mixed-precision, 4-bit bnb quantization, and a cosine learning rate scheduler.

⁵<https://github.com/hiyouga/LLaMA-Factory>

K Evaluation Benchmark Details

This section provides additional details on the benchmarks used in our experiments (Section 5).

BFCL-V3: The BFCL-V3 (Patil et al., 2025) benchmark evaluates multi-turn tool-calling across four challenging categories: *Base*, *Missing Function*, *Missing Parameter*, and *Long Context*. It focuses on conversational scenarios that require clarification, refusal, and robustness. Performance is reported as execution accuracy in a live, executable environment, with credit awarded only when all steps in a dialogue are correctly resolved.

API Bank: The API Bank (Li et al., 2023) benchmark consists of 8 domains, 73 tools, and 314 dialogues. We evaluate on Level 1 (“Call”) and Level 2 (“Retrieve+Call”). As tool retrieval is not the focus of this paper, we provide the set of required tools to the model for both levels. The evaluation is

Table 11: Generation cost per 1,000 dialogues, measured on 2× NVIDIA H100 GPUs serving GPT-OSS-120B via vLLM. In/Out are input/output tokens in millions; Time is end-to-end wall clock, including parallelism. API synthesis is a one-time cost that produces tool definitions reused across all 20 domains. A dash indicates a stage that is not part of the ToolFlow pipeline.

Stage	ToolWeave				ToolFlow			
	Calls	In (M)	Out (M)	Time (h)	Calls	In (M)	Out (M)	Time (h)
API Synthesis	2,074	1.6	0.6	0.2	–	–	–	–
Goal Generation	3,632	2.2	0.7	0.1	–	–	–	–
Plan Generation	1,693	1.2	0.4	< 0.1	950	1.1	0.5	0.2
Dialogue Generation	20,020	46.7	8.1	2.0	12,656	39.0	5.0	2.0
Dialogue Refinement	2,734	4.3	0.7	0.2	–	–	–	–
Total	30,153	56.0	10.5	2.6	13,606	40.1	5.5	2.2

Table 12: Key fine-tuning hyperparameters for all models.

Hyperparameter	Llama-3.1-8B-Instruct	Llama-3.1-70B-Instruct	Phi-4
<i>Model Details</i>			
quantization_bit	4	4	4
flash_attn	fa2	fa2	fa2
<i>LoRA Configuration</i>			
lora_rank	32	8	16
lora_alpha	64	32	32
lora_dropout	0.05	0.05	0.05
lora_target	all	all	all
<i>Training Details</i>			
num_train_epochs	1.0	1.0	1.0
warmup_ratio	0.05	0.1	0.1
per_device_train_batch_size	1	1	1
gradient_accumulation_steps	8	2	8
gradient_checkpointing	false	true	false

conducted in an off-policy, teacher-forced manner, where the model must predict the next step given the correct history.

CONFETTI: The CONFETTI (Alkhouli et al., 2025) benchmark is a multi-turn, multi-step conversational benchmark with 109 human-simulated conversations covering 86 tools. A key characteristic is its explicit inclusion of conversational complexities, such as follow-ups, goal correction, goal switching, ambiguous goals, and over- or under-filled parameters. We focus on evaluating the turn-level tool-calling accuracy.

L Details of Ablation

This section provides additional details on the ablation studies discussed in Section 5.4 of the main paper.

L.1 Detailed Frontier Model Ablation Results

Table 13 provides the comprehensive performance breakdown of Llama-3.1-70B fine-tuned on data generated by synthesis models of varying scales and licensing types. This includes detailed metrics for

Base, Missing Function, Missing Parameter, and Long Context scenarios.

L.2 Detailed Component Ablation Results

This section provides the comprehensive results for the pipeline component ablations discussed in Section 5.4.

Necessity of Dialogue Post-Processing: Table 14 isolates the impact of the post-processing stage. Injecting robustness patterns yields a substantial **+6.26%** absolute gain in Multi-Turn Accuracy. Evaluated on Llama-3.1-8B-Instruct, post-processing produces large and consistent gains across all metrics, particularly in the “Missing Function” category (+7.00).

Necessity of Structured Sampling and Fine-Grained Planning: Table 15 validates the architecture of the generation pipeline itself. To ensure comparability, all settings in this table exclude post-processing.

1. **Impact of Structured Sampler:** Replacing our motif-based sampler with a random walk

Table 13: Full performance breakdown of Llama-3.1-70B fine-tuned on data generated by different synthesis models.

Synthesis Model	Acc	Base	MissFunc	MissParam	LongCtx
ToolWeave (Mistral-Small-24B) [Open]	28.00	29.00	32.50	26.50	24.00
ToolWeave (Mistral-Medium) [Commercial]	28.88	27.50	39.00	30.00	19.00
ToolWeave (GPT-OSS-120B) [Open]	33.25	37.50	35.00	32.00	28.50
ToolWeave (GPT-5-mini) [Frontier]	39.75	42.50	43.00	37.00	36.50

Table 14: Performance impact of the post-processing (postproc) stage on Llama-3.1-8B-Instruct.

Setting	Acc	Base	MissFunc	MissParam	LongCtx
ToolWeave (GPT-OSS seed) w/o postproc	13.62	14.50	14.50	13.50	12.00
ToolWeave (GPT-OSS seed) w/ postproc	19.88	23.00	21.50	15.50	19.50
<i>Absolute Gain</i>	+6.26	+8.50	+7.00	+2.00	+7.50

(Row 2) degrades accuracy from 13.62% to 12.25%.

2. **Impact of Fine-Grained Planner:** Row 3 shows that replacing our planner with ToolFlow’s monolithic planner collapses performance to 7.50%, demonstrating the planner is the most critical component for enabling non-frontier models.

L.3 Generation Hyperparameters: p_{clar} and p_{inject}

We use $p_{clar} = 0.3$ (Section 3.3) and $p_{inject} = 0.7$ (Section 3.4) for all reported results, and discuss both below.

Sensitivity to p_{inject} : Table 16 sweeps p_{inject} for Llama-3.1-8B-Instruct on BFCL-V3 multi-turn. Average accuracy stays within 21.12–23.88 ($\Delta = 2.76$), comparable to the spread in accuracy we observe across checkpoints saved at different training steps within a single run. The per-category scores fluctuate more widely than the average (Base ranges from 21.00 to 33.50) but show no systematic trend across p_{inject} . We therefore treat p_{inject} as a second-order factor that governs exposure to error-recovery patterns rather than core tool-use capability, and fix it at 0.7 across all model families to guarantee sufficient exposure to recovery behaviour within a fixed training budget, rather than tuning it on the evaluation benchmarks.

Choice of p_{clar} : Unlike p_{inject} , varying p_{clar} requires regenerating the full dialogue corpus, making an exhaustive sweep computationally prohibitive. We set $p_{clar} = 0.3$ so that clarification is present but not dominant, as in human-authored conversational benchmarks such as CONFETTI (Alkhoul

et al., 2025), where seek-information is a meaningful but non-dominant pattern (110 of 663 turns). Note that p_{clar} applies per required parameter, so the two rates are not directly comparable. Because p_{clar} governs only which required parameters are elicited in dialogue, rather than the provenance bindings, schema validation, or dependency tracking that enforce correctness, we expect performance to be stable across a reasonable range.

L.4 LLM-as-Judge Evaluation Details

This appendix provides full details of the LLM-as-judge protocol used in Section 4.4 to assess the semantic quality of synthetic dialogues.

Judge Model and Decoding Setup We follow the evaluation protocol of Wang et al. (2025), which uses GPT-4 as its judge, but adopt the more recent GPT-5-mini as our primary judge model; we additionally evaluate with Gemini 2.5 Flash, Claude Opus 4.6, and GPT-5.5 to assess judge sensitivity. All judges are queried through their provider APIs with default decoding settings. Each dialogue is evaluated in a single forward pass; no score aggregation or self-consistency sampling is applied.

Evaluation Dimensions Each dialogue is independently rated along four dimensions on a 1–5 Likert scale:

- **Naturalness:** Fluency and human-likeness of the conversation.
- **Coherence:** Logical consistency and relevance across turns.
- **Helpfulness:** Degree to which the dialogue successfully addresses the user’s goal.
- **Accuracy:** Correctness of tool usage, arguments, and factual content.

Table 15: Performance comparison across pipeline settings (excluding post-processing). Validates the Structured Sampler and Fine-Grained Planner.

Setting	Acc	Base	MissFunc	MissParam	LongCtx
ToolWeave (no postproc)	13.62	14.50	14.50	13.50	12.00
Random sampling + ToolWeave	12.25	13.00	14.00	13.50	8.50
Structured sampling + ToolFlow	7.50	9.50	7.50	5.00	8.00
ToolFlow baseline (GPT-OSS seed)	7.62	11.00	4.50	5.50	9.50

Table 16: Sensitivity to the failure-injection probability p_{inject} on BFCL-V3 multi-turn using Llama-3.1-8B-Instruct. The row $p_{inject} = 0.7$ is the configuration used throughout the paper.

p_{inject}	Avg. Acc.	Base	Miss Func.	Miss Param.	Long Ctx.
0.1	21.25	30.50	16.00	16.00	22.50
0.3	23.75	26.00	23.50	20.50	25.00
0.5	23.38	33.50	16.50	17.00	26.50
0.7	21.12	28.00	18.00	18.50	20.00
0.9	23.88	32.50	16.50	22.00	24.50
1.0	22.12	21.00	23.00	22.00	22.50

Table 17: Dialogue quality under four LLM judges. Nat=Naturalness, Coh=Coherence, Hel=Helpfulness, Acc=Accuracy.

Judge	Dataset	Nat	Coh	Hel	Acc	Avg
GPT-5-mini	ToolWeave	4.38	4.12	3.89	4.38	4.19
	ToolFlow	3.85	2.94	3.21	3.27	3.32
	ToolDial	3.38	3.42	3.00	3.82	3.41
Gemini 2.5 Flash	ToolWeave	3.38	3.45	3.74	3.68	3.56
	ToolFlow	2.69	1.50	2.13	1.96	2.07
	ToolDial	2.69	2.65	2.87	3.75	2.99
Claude Opus 4.6	ToolWeave	2.87	3.33	3.08	3.44	3.18
	ToolFlow	2.34	1.75	2.22	2.37	2.17
	ToolDial	2.38	2.75	2.71	3.23	2.77
GPT-5.5	ToolWeave	2.86	3.07	3.04	3.42	3.10
	ToolFlow	1.98	1.06	1.35	1.38	1.44
	ToolDial	2.58	2.82	2.50	3.30	2.80

Sampling Protocol For each dataset, we randomly sample 200 dialogues, stratified uniformly across the available domains (20 domains for ToolWeave and ToolFlow). ToolDial dialogues are sampled from the publicly released dataset and may not cover all domains. Each sampled dialogue is evaluated exactly once.

Judge Robustness To assess judge sensitivity, we repeat the protocol with three additional judge models (Gemini 2.5 Flash, Claude Opus 4.6, and GPT-5.5) in addition to GPT-5-mini. Table 17 shows that ToolWeave outperforms ToolFlow and ToolDial across judges, although absolute scores vary considerably across models. This indicates that our conclusions are robust to the choice of judge model, and the improvements are not an artifact of a specific evaluator.

Evaluation Prompt The judge is provided with a full dialogue transcript and instructed to score it independently along the four dimensions above. Dataset and model identities are not disclosed to the judge. We adopt the full evaluation prompt from Wang et al. (2025), which is shown below:

LLM-as-judge Evaluation Prompt

You are asked to evaluate some synthetic dialogue data. These synthetic dialogue occur between the user, the AI assistant, and the tool. Please evaluate the data based on the following criteria, assigning a score from 1 to 5 for each category. Use the detailed descriptions below to guide your assessment:

- Naturalness (1-5 points): Only evaluate whether the user's request and response is natural and realistic. Focus more on the natural flow of the conversation and less on the choice of words. For example, pay attention to whether users will ask similar questions in real scenarios. And assess whether user behavior is natural. For example, real users rarely ask similar questions consecutively or ask longer questions.
- Coherence (1-5 points): Evaluate the overall flow and logical connection between the turns in the conversation. Focus on checking whether the user's previous and subsequent rounds of requests are relevant. Also, ensure that meaningful order of agent interactions is maintained. For example, the agent should not provide information before the user requests it. Similarly, the relevant tools must be invoked only after the assistant calls the tool with the relevant parameters. Such discrepancies should be heavily penalized.
- Helpfulness (1-5 points): Determine the effectiveness and value of the AI assistant's responses in addressing the user's needs.
- Accuracy (1-5 points): Check for the accuracy and consistency of the information provided. Everything returned by the tool is assumed to be accurate. However, at the same time, if the assistant hallucinates or makes up information, it should be heavily penalized.

Below are some examples that are rated poorly in each category:

1. Naturalness:

- User: "Can you tell me about the weather?"
- Assistant: "The weather is nice."
- User: "What about tomorrow?"
- Assistant: "Tomorrow will be nice too."

2. Coherence:

- User: "What's the capital of France?"
- Assistant: "The capital of France is Paris."
- User: "And the capital of Germany?"
- Assistant: "The capital of Germany is Berlin."

3. Helpfulness:

- User: "I need help with my homework."
- Assistant: "What subject is it?"
- User: "Math."
- Assistant: "Okay."

4. Accuracy:

- User: "What's the capital of Italy?"
- Assistant: "The capital of Italy is Rome."
- User: "What's the capital of Spain?"
- Assistant: "The capital of Spain is Barcelona."

Some more examples specifically catered to tool usage:

1. Coherence: (Tool simulated before assistant provides tool to be called along with its input parameters)

- User: "Can you raise a ticket for my internet issue?"
- Tool: {"ticket_id": "12345"}

2. Coherence: (Assistant goes on rambling trying to correct erroneous tool output but never calls the tool)

- User: "Please book a flight for me."
- Assistant: "Sure, I can help you with that. Can you please provide me with the details of your flight?"
- User: "I need a flight from New York to San Francisco on March 10th."
- Assistant: book_flight({"from": "New York", "to": "San Francisco", "date": "March 10th"})
- Tool: "Flight ID not found."
- Assistant: "I apologize for the inconvenience. It seems that there was an error with the flight ID. Let me try again."
- Tool: "Flight ID not found."

3. Coherence: (Assistant talks about calling tool but does not use the required tool-calling format)

- User: "Can you check the weather for me?"
- Assistant: "Sure, I can help you with that. Let me check the weather for you."
- Tool: {"temperature": "72°F", "condition": "Sunny"}

- Assistant: "The weather is 72°F and sunny."

4. Accuracy: (Assistant hallucinates information)

- User: "Can you reserve a table for two at an Italian restaurant in New York City for tomorrow night?"

- Assistant: "Sure, I have reserved a table for two at 'Luigi's Italian Bistro' in New York City for tomorrow night at 7 PM."

All such examples must be heavily penalized, and the scores they receive should be 1 or 2.

Please use a more ****CRITICAL**** and ****STRICT**** evaluation method. After scoring, please provide brief comments or feedback for each category to explain your ratings.

Please provide your evaluation in the following format:

Evaluation of Synthetic Dialogue Data

1. Naturalness: [Score] / 5

- Comments: [Brief comments or feedback]

2. Coherence: [Score] / 5

- Comments: [Brief comments or feedback]

3. Helpfulness: [Score] / 5

- Comments: [Brief comments or feedback]

4. Accuracy: [Score] / 5

- Comments: [Brief comments or feedback]

M ToolWeave Sample Data - Goal, Plan, Tools, and Dialogue

We present a detailed example from the customer support domain that illustrates a multi-step dialogue. The example shows the three tools used in this dialogue, the high-level goal constructed over them, the nine-step fine-grained plan, and the multi-turn dialogue that realizes the plan. Every argument in the executed tool calls is traceable to the plan: `customer_id` is bound to the `user_id` returned by `get_ticket_details`, and `feedback_record_id` to the `feedback_id` returned by `submit_feedback`.

M.1 Tools List

Tool 1: `get_ticket_details`

```
{
  "name": "get_ticket_details",
  "description": "Retrieves comprehensive data for a given support ticket, encompassing its
  ↳ full record, current status, and details of the personnel or team to which it is
  ↳ assigned.",
  "parameters": {
    "type": "object",
    "properties": {
      "support_ticket_id": {"type": "string", "description": "Unique identifier of the
      ↳ support ticket whose full details are being requested."}
    },
    "required": ["support_ticket_id"]
  },
  "results": {
    "type": "object",
    "properties": {
      "user_id": {"type": "string", "description": "Identifier of the user who opened the
      ↳ ticket."},
      "subject": {"type": "string", "description": "Title of the ticket."},
      "description": {"type": "string", "description": "Full description provided by the
      ↳ user."},
      "priority": {"type": "string", "enum": ["low", "medium", "high"], "description":
      ↳ "Urgency level assigned to the ticket."},
      "status": {"type": "string", "enum": ["open", "pending", "resolved", "closed"],
      ↳ "description": "Current lifecycle state of the ticket."},
      "channel": {"type": "string", "enum": ["email", "phone", "chat"], "description": "How
      ↳ the ticket was originally submitted."},
      "creation_timestamp": {"type": "string", "format": "date-time", "description": "When
      ↳ the ticket was first created."},
      "last_updated_timestamp": {"type": "string", "format": "date-time", "description":
      ↳ "Most recent time any field on the ticket changed."},
      "assigned_agent_id": {"type": "string", "description": "Identifier of the support
      ↳ agent currently handling the ticket, if any."}
    }
  }
}
```

Tool 2: `submit_feedback`

```
{
  "name": "submit_feedback",
  "description": "Gather and store feedback submitted by a user after a support interaction,
  ↳ specifically related to a particular support ticket, including satisfaction ratings,
  ↳ comments, and any additional observations about the service experience.",
  "parameters": {
    "type": "object",
    "properties": {
      "rating": {"type": "string", "enum": ["1", "2", "3", "4", "5"], "description":
      ↳ "Customer satisfaction rating (1 = very dissatisfied, 5 = very satisfied)."},
      "comments": {"type": "string", "description": "Optional free-text comments elaborating
      ↳ on the rating."},
    }
  }
}
```

```

    "support_ticket_id": {"type": "string", "description": "Unique identifier of the
    ↪ support ticket for which the feedback is being recorded."},
    "customer_id": {"type": "string", "description": "Unique identifier of the customer
    ↪ who is submitting the feedback."}
  },
  "required": ["rating", "support_ticket_id", "customer_id"]
},
"results": {
  "type": "object",
  "properties": {
    "feedback_id": {"type": "string", "description": "System-generated identifier for the
    ↪ feedback record."},
    "submitted_at": {"type": "string", "format": "date-time", "description": "Timestamp
    ↪ when the feedback was received."},
    "status": {"type": "string", "enum": ["received", "processed"], "description":
    ↪ "Current processing state of the feedback."}
  }
}
}

```

Tool 3: trigger_feedback_followup

```

{
  "name": "trigger_feedback_followup",
  "description": "Schedules a follow-up interaction with a user triggered by the feedback
  ↪ they have submitted, specifically targeting low-rating responses, to address concerns,
  ↪ gather additional insights, and ensure the feedback loop is fully closed.",
  "parameters": {
    "type": "object",
    "properties": {
      "followup_method": {"type": "string", "description": "Channel to use for the
      ↪ follow-up.", "enum": ["email", "phone", "chat"]},
      "preferred_time": {"type": "string", "format": "date-time", "description": "When the
      ↪ user would like to be contacted (optional)."},
      "feedback_record_id": {"type": "string", "description": "Unique identifier of the
      ↪ feedback entry returned by the submit_feedback operation."}
    },
    "required": ["followup_method", "feedback_record_id"]
  },
  "results": {
    "type": "object",
    "properties": {
      "followup_id": {"type": "string", "description": "System-generated identifier for the
      ↪ scheduled follow-up."},
      "scheduled_at": {"type": "string", "format": "date-time", "description": "Timestamp
      ↪ when the follow-up is set to occur."},
      "status": {"type": "string", "description": "Current state of the follow-up request.",
      ↪ "enum": ["scheduled", "cancelled", "completed"]}
    }
  }
}

```

M.2 High Level Goal

Goal

Retrieve the full details and assignment of a specific support ticket, record the user's satisfaction ratings and comments about the interaction, and trigger a follow-up outreach when feedback is low to address concerns and close the feedback loop.

M.3 Plan

Multi-step Plan

Step 1 (USER_UTTERANCE)

Utterance: *Please retrieve the full record for the specified support ticket, including its current status and which person or team it's assigned to.*

Provided params:

```
{"get_ticket_details.support_ticket_id":  
  ↳ "$user_provided_$get_ticket_details.support_ticket_id"}
```

Step 2 (CALL_TOOL: get_ticket_details)

Parameters:

```
{"get_ticket_details.support_ticket_id":  
  ↳ "$user_provided_$get_ticket_details.support_ticket_id"}
```

Step 3 (ASSISTANT_RESPONSE_TOOL)

Summarizes tools:

```
["get_ticket_details"]
```

Outputs:

```
["get_ticket_details.user_id", "get_ticket_details.subject",  
  ↳ "get_ticket_details.description", "get_ticket_details.priority",  
  ↳ "get_ticket_details.status", "get_ticket_details.channel",  
  ↳ "get_ticket_details.creation_timestamp", "get_ticket_details.last_updated_timestamp",  
  ↳ "get_ticket_details.assigned_agent_id"]
```

Step 4 (USER_UTTERANCE)

Utterance: *Okay, now record and store my satisfaction rating and comments for that ticket, and if the feedback is low please trigger a follow-up outreach to address the concerns and close the feedback loop.*

Provided params:

```
{"submit_feedback.support_ticket_id": "$user_provided_$submit_feedback.support_ticket_id"}
```

Step 5 (ASSISTANT_CLARIFICATION)

Parameter names:

```
["submit_feedback.rating", "submit_feedback.comments",  
  ↳ "trigger_feedback_followup.followup_method"]
```

Step 6 (USER_RESPONSE_TO_CLARIFICATION)

Provides params:

```
{"submit_feedback.rating": "$user_provided_$submit_feedback.rating",  
  ↳ "submit_feedback.comments": "$user_provided_$submit_feedback.comments",  
  ↳ "trigger_feedback_followup.followup_method":  
  ↳ "$user_provided_$trigger_feedback_followup.followup_method"}
```

Step 7 (CALL_TOOL: submit_feedback)

Parameters:

```
{ "submit_feedback.rating": "$user_provided_$submit_feedback.rating",  
  ↪ "submit_feedback.comments": "$user_provided_$submit_feedback.comments",  
  ↪ "submit_feedback.support_ticket_id":  
  ↪ "$user_provided_$submit_feedback.support_ticket_id", "submit_feedback.customer_id":  
  ↪ "$get_ticket_details.user_id" }
```

Step 8 (CALL_TOOL: trigger_feedback_followup)

Parameters:

```
{ "trigger_feedback_followup.followup_method":  
  ↪ "$user_provided_$trigger_feedback_followup.followup_method",  
  ↪ "trigger_feedback_followup.feedback_record_id": "$submit_feedback.feedback_id" }
```

Step 9 (ASSISTANT_RESPONSE_TOOL)

Summarizes tools:

```
[ "submit_feedback", "trigger_feedback_followup" ]
```

Outputs:

```
[ "submit_feedback.feedback_id", "submit_feedback.submitted_at", "submit_feedback.status",  
  ↪ "trigger_feedback_followup.followup_id", "trigger_feedback_followup.scheduled_at",  
  ↪ "trigger_feedback_followup.status" ]
```

M.4 Dialogue

Multi-turn Dialogue

System: Current time: 2025-11-21 21:59:31.

User: Please retrieve the full record for support ticket ID ST-20251118-8392, and show me the complete record/history including its current status and which person or team it's assigned to.

Assistant:

```
get_ticket_details({ "support_ticket_id": "ST-20251118-8392" })
```

Tool:

```
{"user_id": "U-54892", "subject": "Unable to access payroll portal - receiving 403 error", "description": "User reports inability to access the company's payroll portal since 2025-11-18. When attempting to log in, the site returns a 403 Forbidden error. User has tried multiple browsers (Chrome, Firefox), cleared cache/cookies, attempted access from a different network and device, and confirmed credentials are correct. Screenshot of the error page and browser console logs were attached to the report. This impacts payroll submission for the current pay period; user indicated a hard deadline for submissions on 2025-11-22. Ticket was initially triaged and escalated to engineering for possible permissions or firewall issue.", "priority": "high", "status": "pending", "channel": "email", "creation_timestamp": "2025-11-18T09:14:22Z", "last_updated_timestamp": "2025-11-21T19:45:10Z", "assigned_agent_id": "AG-2384"}
```

Assistant: Here's the full record for support ticket ST-20251118-8392: user_id is U-54892; subject is "Unable to access payroll portal - receiving 403 error"; description: "User reports inability to access the company's payroll portal since 2025-11-18. When attempting to log in, the site returns a 403 Forbidden error. User has tried multiple browsers (Chrome, Firefox), cleared cache/cookies, attempted access from a different network and device, and confirmed credentials are correct. Screenshot of the error page and browser console logs were attached to the report. This impacts payroll submission for the current pay period; user indicated a hard deadline for submissions on 2025-11-22. Ticket was initially triaged and escalated to engineering for possible permissions or firewall issue."; priority: high; status: pending; channel: email; creation_timestamp: 2025-11-18T09:14:22Z; last_updated_timestamp: 2025-11-21T19:45:10Z; assigned_agent_id: AG-2384.

User: Current time: 2025-11-21 21:59:31. Please record feedback for support ticket ST-20251118-8392. I want to submit a satisfaction rating and written comments for that ticket — prompt me for the rating and the comment text if you need them, store the feedback, and if the rating is low please arrange a follow-up outreach (email or phone) to address the concerns and close the feedback loop.

Assistant: For ticket ST-20251118-8392, what satisfaction rating would you like to submit (choose 1 = very dissatisfied through 5 = very satisfied), what comment would you like to include, and which follow-up method should we use if the rating is low — email, phone, or chat?

User: Current time: 2025-11-21 21:59:31. Please record a satisfaction rating of 2 with the comment "The issue is still unresolved." and arrange follow-up by phone.

Assistant:

```
submit_feedback({"rating": "2", "comments": "The issue is still unresolved.", "support_ticket_id": "ST-20251118-8392", "customer_id": "U-54892"})
```

Tool:

```
{"feedback_id": "FB-20251121-68317", "submitted_at": "2025-11-21T21:59:45Z", "status": "received"}
```

Assistant:

```
trigger_feedback_followup({"followup_method": "phone", "feedback_record_id": "FB-20251121-68317"})
```

Tool:

```
{"followup_id": "FUP-FB-20251121-68317-01", "scheduled_at": "2025-11-22T09:30:00Z", "status": "scheduled"}
```

Assistant: I recorded your satisfaction rating of 2 and comment "The issue is still unresolved." for ticket ST-20251118-8392: the feedback was saved with ID FB-20251121-68317 at 2025-11-21T21:59:45Z and its status is received, and a phone follow-up has been scheduled (follow-up ID FUP-FB-20251121-68317-01) for 2025-11-22T09:30:00Z with status scheduled.

N Example Dialogue Generated via ToolFlow

This example dialogue highlights a common issue in dialogues generated via the ToolFlow pipeline, where the tool agent hallucinates a tool response without the assistant agent providing an actual tool call, resulting in a fabricated tool reply (as seen in the first assistant response and the first tool response).

Multi-turn Dialogue

User: I've just bought a new field and have the map outline. Could you define its boundary with this GeoJSON: `{"type": "Feature", "geometry": {"type": "Polygon", "coordinates": [[[30.0, 10.0], [40.0, 40.0], [20.0, 40.0], [10.0, 20.0], [30.0, 10.0]]]}, "properties": {}}`? The field ID is "field-12345".

Assistant: Sure, defining the boundary for field field-12345 now.

Tool:

```
{"field_boundary_id": "boundary-67890", "status": "defined"}
```

Assistant: Great! The boundary for field field-12345 has been successfully defined (status: defined). I can now pull a short-term weather forecast for the next week for this field. Would you like me to retrieve that for you?

User: Yes, please pull the short-term weather forecast for the next week for field-12345.

Assistant: Sure! To get the short-term weather forecast I'll need the farm identifier that corresponds to this field. Could you provide the farm ID (or let me know if the field ID can be used in place of a farm ID)?

User: Sure, the farm ID is farm-abc123.

Assistant:

```
get_weather_forecast({"farm_identifier": "farm-abc123"})
```

Tool:

```
{"temperature_celsius": 22.3, "precipitation_mm": 0.0, "wind_speed_kph": 14.5, "condition": "sunny"}
```

Assistant: Here's the forecast I've retrieved for tomorrow at your field field-12345 (farm-abc123):

O Some Example Dialogues From ToolDial Training Set

The following examples illustrate some of the issues why LLMs fine-tuned on ToolDial data exhibit inferior performance on the BFCL-V3 benchmark (see Table 5).

Example O.1 shows that models trained on such data struggle with parameter elicitation and error recovery, the assistant neither asks for the missing hotel IP address nor attempts to resolve the misunderstanding after the user's correction.

O.1 Example 1

Multi-turn Dialogue

User: Can you help me find detailed information about local businesses, such as restaurants and cafes, near my hotel? I only have the hotel's IP address and need to find its geo-location first to get the coordinates.

Assistant: I can help you find detailed information about local businesses near your hotel. Would you like me to proceed with finding nearby businesses?

User: No, that's not what I need.

Assistant: I'm sorry, but I cannot fulfill your request due to insufficient information. Please provide more details or try a different query.

O.2 Example 2

Example O.2 illustrates two issues. The user's initial request is somewhat unrealistic and unlikely to occur in a real-world setting. The assistant's clarification questions are phrased using internal parameter names rather than natural, intuitive language that a user would expect.

Multi-turn Dialogue

User: What is the population within a defined area around a city, given only the city's name and its country?

Assistant: Would it be appropriate to retrieve the population data within a defined area around the city you mentioned?

User: That would be great. Sure, that sounds good.

Assistant: Please tell me the radius of the circular area around the coordinates and the coordinates of the geographic point to search nearby.

User: I can provide the radius, which is 300, but I don't know the coordinates.

Assistant: Please provide the namePrefixDefaultLangResults for the city.

User: Sure! The namePrefixDefaultLangResults for the city is San Francisco.

Assistant:

```
[{ "name": "cities", "arguments": { "namePrefixDefaultLangResults":  
  "San Francisco" } } ]
```

Tool:

```
{ "error": "", "response": { "data": [{ "id": 5392171, "wikiDataId": "Q123456", "type": "CITY", "city": "San Francisco", "name": "San Francisco", "country": "United States", "countryCode": "US", "region": "California", "regionCode": "CA", "regionWdId": "Q99", "population": 883305, "coordinates": "[37.7749, -122.4194]"}, { "id": 5902958, "wikiDataId": "Q123457", "type": "CITY", "city": "San Jose", "name": "San Jose", "country": "United States", "countryCode": "US", "region": "California", "regionCode": "CA", "regionWdId": "Q99", "population": 1035317, "coordinates": "[37.3382, -121.8863]"}, { "id": 5322760, "wikiDataId": "Q123458", "type": "CITY", "city": "San Diego", "name": "San Diego", "country": "United States", "countryCode": "US", "region": "California", "regionCode": "CA", "regionWdId": "Q99", "population": 1423851, "coordinates": "[32.7157, -117.1611]"}], "links": { "next": "/v1/geo/cities?offset=60&limit=50" }, "metadata": { "currentOffset": 10, "totalCount": 150 } } }
```

Assistant: Please choose one of the following cities: San Francisco, San Jose, San Diego.

User: I choose San Francisco.

Assistant:

```
get_population({ "radius": "300", "coordinate": "[37.7749, -122.4194]" })
```

Tool:

```
{ "error": "", "response": { "radius": 300, "population": 12000, "description": "The estimated resident population within a 300-meter radius of the specified coordinates in San Francisco is approximately 12,000 individuals. This figure is based on the latest demographic data and reflects the density of the area, which includes residential buildings, parks, and commercial establishments.", "coordinates": "[37.7749, -122.4194]" } }
```

Assistant: The estimated resident population within a 300-meter radius of the specified coordinates in San Francisco is approximately 12,000 individuals. This figure is based on the latest demographic data and reflects the density of the area, which includes residential buildings, parks, and commercial establishments.

User: Thank you for the information. Goodbye!

Assistant: You're welcome! Goodbye!