SCALING LAWS FOR TOOL AGENTS VIA CONTROLLABLE ENVIRONMENT SYNTHESIS

Anonymous authors
Paper under double-blind review

ABSTRACT

We explore how to build performant large language model (LLM) tool-use agents that complete complex, multi-step tasks. Specifically, we study how data composition in synthetically generated tool-use environments affects the training of such agents. To do so, we propose the SynthAPI pipeline, which explicitly controls the diversity of generated environments along the axes of tools, tasks, and the databases on which the tools operate. By systematically varying SynthAPI's generation controls across 849 trained models of four different sizes, we derive a scaling law that predicts downstream tool-agent generalization based on the scale and composition of the synthesized environments. These predictable relationships reveal several lessons for scaling tool-agent data. As data budgets grow, scaling tool and database diversity becomes more effective than increasing the number of tasks, and increasing data complexity yields larger gains. Scaling according to these findings yields agents that outperform models trained on data from prior work in large-scale tool-agent and synthetic environment generation across four evaluation benchmarks.

1 INTRODUCTION

Large language models (LLMs) have demonstrated impressive capabilities as tool-use agents that interact with complex, stateful environments (Trivedi et al., 2024; Patil et al., 2025). These agents must select and compose tools over multiple steps while tracking state changes and generalizing to new environments with new APIs. Progress in developing such agents depends on scaling high-quality, diverse training data. Yet producing this data is challenging because multi-step tool interactions are scarce in internet text and require realistic tasks, diverse tools, and rich environment data.

Prior work addresses this challenge by using LLMs to synthesize tool-use data. While many existing works focus on synthesizing tool-use data to train stateless, read-only tool agents (Schick et al., 2023), work on scaling supervision for stateful tool agents is more limited. Some approaches use LLMs to synthesize entire tool-use trajectories, including simulating the tool responses (Liu et al., 2024a; Tang et al., 2023). These techniques have demonstrated impressive scaling (NVIDIA, 2025; Lee et al., 2026), but complex tool outputs and state transitions can be difficult to simulate reliably. Other approaches synthesize code-backed executable tool environments (Wang et al., 2026; Song et al., 2026; Xu et al., 2026; Dong et al., 2026). However, these efforts primarily emphasize increasing data quantity, leaving open which properties of the generated data drive downstream performance. Tool-agent data varies in the diversity of available tools, the tasks they support, and the database states over which they operate. Understanding how these properties affect generalization is necessary to determine what data to generate as training scales.

To study these relationships, we introduce SynthAPI, a controllable synthesis pipeline for tool-agent environments. SynthAPI uses LLMs to implement apps comprising Python tools that read and modify structured databases, then generates tasks with initial database states (which we call data profiles) and executable success verifiers. The key distinction from prior executable environment synthesis approaches is that SynthAPI provides explicit controls over the diversity of tools, tasks, and starting database states. These controls allow us to vary the composition of the synthesized training tasks and study how a fixed budget should be allocated among new tools, new tasks, and different data profiles. Figure 1 summarizes the pipeline and its controls.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

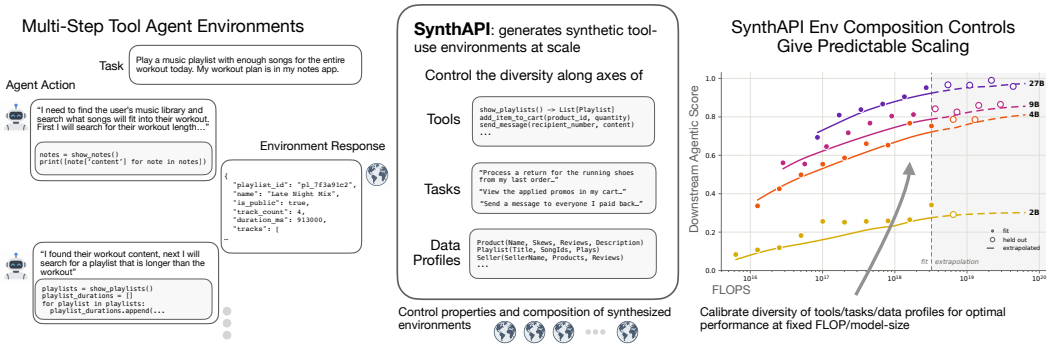


Figure 1: Overview of SynthAPI. On the left, we show the tool-agent tasks we focus on, where given an instruction, an agent must reason and execute tool calls to complete the task. In the middle, the SynthAPI pipeline generates synthetic tool-agent environments at scale. SynthAPI generates this data while controlling the diversity of the tools, tasks, and starting database states. These controls allow deriving the scaling laws on the right (presented in detail in Figure 3).

We use this controllability to derive a predictive scaling law that uses data composition factors to predict downstream tool-agent generalization. We find that the data composition that best supports downstream generalization changes with scale. Comparisons at fixed task budgets show that, as these budgets grow, expanding tool diversity becomes more valuable than generating additional tasks within the same set of tools. Furthermore, repeating the same tasks with different data profiles also becomes more important than generating distinct tasks. Training with more complex data profiles also yields larger gains at higher budgets. These findings emphasize the importance of tool and initial database state diversity over simply increasing the task count. Our joint scaling law captures these effects through a composition-dependent performance ceiling, with the number of tasks per app controlling progress toward that ceiling. Across four model sizes, we hold out the runs in the top 25% of training FLOPs and show that the law extrapolates to them, including runs with up to $16\times$ the largest FLOP budget used for fitting, achieving a mean absolute error of 0.0285 on the normalized downstream score, compared with 0.1706 for a model based on task quantity alone.

Our experiments show that these findings translate into stronger performance on AppWorld, BFCL, and ASTRA-Bench (Trivedi et al., 2024; Patil et al., 2025; Xiu et al., 2026). At a fixed task budget, the recipe identified by our analysis outperforms a suboptimal recipe across all model sizes and benchmarks. We scale up the SynthAPI data and training and find that the resulting models outperform models trained on prior large-scale tool-agent data (NVIDIA, 2025) by 30.5 percentage points on average across the three benchmarks and the 4B and 9B model sizes. We also compare to strong per-benchmark baselines, outperforming the strongest baseline at each model size by 7.6–11.6 percentage points on AppWorld Test-Challenge and 2.0–3.7 points on BFCL (Lee et al., 2026; Xu et al., 2026; Song et al., 2026). These results demonstrate the practical value of our findings on data composition and predictive scaling for improving tool-agent performance.

2 Related Work

Synthetic supervision has been widely used to train LLMs for tool use. Early methods train on demonstrations over predefined API collections (Schick et al., 2023; Qin et al., 2023; Chen et al., 2024), while subsequent work generates supervision through live APIs and executable functions (Liu et al., 2024b; Prabhakar et al., 2025). Other approaches simulate API responses with LLMs (Liu et al., 2024a; Tang et al., 2023) or combine data from multiple sources and synthesis pipelines (Zhang et al., 2025). DIVE (Chen et al., 2026) analyzes tool diversity in evidence-based question answering, whereas SynthAPI studies multi-step read-write workflows over structured databases, a setting emphasized by recent tool-agent benchmarks (Trivedi et al., 2024; Patil et al., 2025). Nemotron-Agentic (NVIDIA, 2025) scales LLM-simulated tool interactions to 335k trajectories. SimSAT (Lee et al., 2026) extends such simulation to long-horizon tasks. However, these simulators rely on an LLM to generate every tool response, making increasingly complex outputs and state

transitions difficult to simulate reliably. Thus, unlike these works, SynthAPI scales tool data through database-backed executable environments.

Like SynthAPI, recent work also synthesizes executable tool-use environments backed by persistent databases (Wang et al., 2026; Song et al., 2026; Xu et al., 2026; Dong et al., 2026). Unlike these works, SynthAPI couples this synthesis with explicit controls over the diversity and complexity of tools, task instructions, and initial database states. These controls let us disentangle how training-data composition affects downstream generalization. Our analysis goes beyond prior scaling studies that primarily vary the number of synthesized environments or examine domain diversity at a fixed task count (Tu et al., 2026). Specifically, we study how to allocate a fixed data budget among new tools, additional tasks for existing tools, and alternative database states for existing tasks. We further derive a joint scaling law that extrapolates aggregate downstream generalization to larger compute budgets and predicts which data allocations maximize performance. We use these predictions to design a training recipe that improves agent performance over prior synthetic-data generation approaches.

Scaling laws characterize predictable relationships between model quality, model size, data volume, and training compute. Early work established power-law behavior in language-model pre-training, while compute-optimal analyses studied how a fixed budget should be divided between parameters and tokens (Kaplan et al., 2020; Hoffmann et al., 2022). Related studies extend scaling analysis to transfer learning, fine-tuning, and reinforcement-learning post-training (Hernandez et al., 2021; Tan et al., 2025). For tool-use models, prior work develops scaling laws for pre-training with agent data, using benchmark loss as the prediction target (Zhuang et al., 2025). In contrast, SynthAPI predicts downstream task success from explicit environment-generation factors, relating generalization to the scale, diversity, and complexity of generated tools, tasks, and databases.

3 SynthAPI: Synthetic Environments for Tool Agents

SynthAPI is a controllable pipeline for synthesizing interactive tool-agent environments. SynthAPI consists of two stages where first apps consisting of APIs and databases are coded in Python and next tasks are constructed using the functionality in these apps.

3.1 API Generation

In the first phase of creating synthetic tool-agent tasks, SynthAPI generates a wide diversity of tools to base tasks on. SynthAPI does this by synthesizing "apps", which are a coherent collection of interacting tools that can be composed to solve realistic tasks. This organization naturally produces dependencies among tools and mirrors deployment settings in which agents use tools to accomplish user goals. These apps are each independently implemented in Python code from scratch by an LLM, which we term the "builder LLM". We use Gemini-3.1 Pro (Google DeepMind, 2026) with internet access disabled as the builder LLM. We manually define a broad app taxonomy consisting of shopping, music, productivity, finance, fitness, messaging, utilities, and social. The builder LLM starts to generate an app by generating "metadata" consisting of the app name, a short app description, the taxonomy category the app belongs to, and a list of functionalities. The functionalities are short descriptions of interactions possible in the app, such as "Initiate a return" or "Apply a promo code to your shopping cart" in the case of a shopping app.

Next, SynthAPI generates a database schema for the app from the metadata to ground app functionality in complex stateful interactions. For example, in a shopping app there need to be products, inventory, users, addresses, payment methods, reviews, carts, and orders for the tools in the app to implement meaningful behaviors. Given the app metadata, the builder LLM generates a `db.py` file that contains Python `Enum` and `dataclass` definitions defining all data types necessary to instantiate the provided app functionalities. These classes only specify data formats rather than instantiated database states. The final step of app creation is generating an API layer that operates on the database schema. From the contents of `db.py` and the app metadata, the builder LLM generates an `apis.py` file for the app. The API file consists of a set of Python functions defining the possible operations in the app. Each function takes as input a database object that tracks instantiated data schema objects, along with any other input arguments, which may consist of the defined data types. These functions read or modify instantiated database objects and collectively define the

agent's available tools. In the shopping example, the resulting APIs may search and filter products, inspect reviews, manage a favorites list or cart, create an order, process a return, and update inventory. There are both private helper functions and public functions that are exposed to the agent as the tool set for this environment.

In summary, in the first phase, SynthAPI generates a set of apps, each of which contains app metadata, a data schema Python file, and an API definition file. All components are generated from scratch by the builder LLM. See Section A.1 for all details and prompts on app generation.

3.2 Task Generation

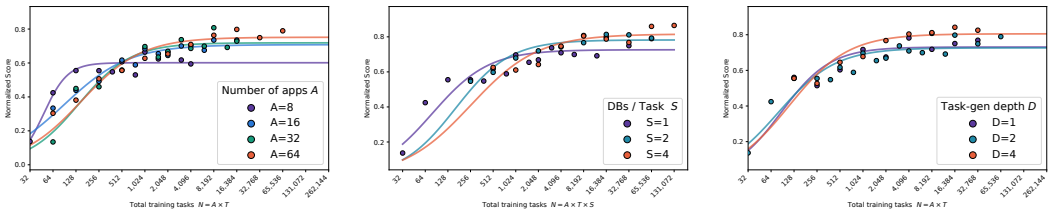
The apps specify which tools and data schemas are possible for the agent to interact with, but they do not specify actual tasks to complete. The second phase of SynthAPI therefore synthesizes actionable tasks for agents to complete using the provided APIs. A task is defined as a tuple $T = \big(x, v, \mathrm{Db}^0, \pi^*\big)$, where x is the task instruction, Db^0 is the starting database state, v is verifier code that determines whether the task has been completed, and π^* is an oracle solver program which solves the task from the starting state. Next, we detail these components and how they are generated.

First, SynthAPI generates the task instruction by sampling an app and subsampling F functionalities from its metadata functionality list. The builder LLM is then prompted to generate n task instructions that utilize these functionalities. To increase instruction diversity, previously generated instructions are included as context for subsequent generations. The instructions contain no ambiguity about the goal and require no additional information or interaction with an external user.

Next, given the task instruction x, SynthAPI generates the verifier v. The verifier is a Python function that takes as input the database state along with an optional answer string provided by the agent and outputs a binary indicator of whether the task instruction has been successfully completed in the database state. The verifier can access historical information from the database state since the database contains the full historical record for many types of information, such as an order history for a shopping app, and the database also logs the sequence of state changes in the episode. The builder LLM generates the verifier from the instruction and all the app data class and API definitions.

Provided (x, v) and the app's `db.py`, SynthAPI then prompts the builder LLM to generate the initial database state Db^0 which consists of instantiated data objects from the app schema. A first pass populates the data objects with the minimum data setup $\mathrm{Db}^0_{\text{core}}$ required to make the task feasible. Next, the data generator creates a batch of extra distractor data Db^0_1 to make the task more complex and realistic. $D-1$ such extra batches of data are created, thus creating D total batches of data $\mathrm{Db}^0 = \mathrm{Db}^0_{\text{core}} \cup \mathrm{Db}^0_1 \cup \cdots \cup \mathrm{Db}^0_{D-1}$. From (x, v, Db^0), the builder LLM generates an oracle solver π^* to check that the task is feasible to complete. The oracle solver is a Python code block that executes the same APIs available to the agent to complete the instruction. π^* is generated using the entire database object Db^0, and thus it can directly access privileged identifiers and information, unlike the agent, which needs to query the database and tools to discover the right information. Finally, SynthAPI runs a filtering process and rejects tasks that do not satisfy $v(\mathrm{Db}^0) = 0$ and $v\big(\mathrm{Exec}(\pi^*, \mathrm{Db}^0)\big) = 1$. See Section A.2 for all details and prompts on task generation.

Each task T_i is implemented in the standard Gymnasium environment format (Towers et al., 2024). The agent interacts with the environment by sending Python code blocks. This allows the agent to call multiple tools in one turn and to use loops, conditionals, and other programmatic operations over tool outputs. Tool results are returned to the agent in JSON format, as is standard in other benchmarks. SynthAPI exposes a wide variety of tools, and the tools required for a given task are not known in advance. Rather than placing every tool definition in the initial context, we give the agent a small set of meta-tools for discovering and using the available interface. `list_apps` returns the available apps, `list_tools` returns all tools within a selected app, and `get_tool_documentation` returns a tool's description, input and output arguments, types, and default values. We expose all tools from the app the task is based on, and we also include all tools from 3-5 distractor apps. Finally, the agent must call `complete_task` when it thinks the task is finished, along with an optional answer if the task requires an answer. This discovery-based interface lets the agent load only the tools relevant to the current task, reducing context length while allowing it to transfer its learned behaviors to new tool sets across downstream benchmarks. All tasks involve only tools in a single app and we defer generating multi-app tasks to future work. However, we



(a) App vs. task allocation (b) Database breadth (c) Database depth

Figure 2: Effect of app-to-task allocation, database breadth, and database depth on the 9B student. The x-axis shows the number of tasks N, and the y-axis shows aggregate downstream performance. **(a)** Curves correspond to different allocations of A and T at a fixed total task count $N = AT$; at a fixed value of N, the highest curve identifies the best allocation. **(b)** Curves correspond to different numbers of database instantiations S per task. **(c)** Curves compare minimal task state ($D = 1$), moderate distractor data ($D = 2$), and a larger distractor set ($D = 4$).

show agents trained with SynthAPI generalize to multi-app tasks in downstream evaluations. See Section A.3 for full details on the SynthAPI environment interface.

Ultimately, SynthAPI creates environments consisting of tools $\mathcal{A} = \{a_1, \ldots, a_A\}$ and tasks $\mathcal{T} = \{T_1, \ldots, T_N\}$ within these apps. The key contribution of SynthAPI is controllability of the composition and properties of the task set $(\mathcal{A}, \mathcal{T})$ via the following controls:

- A **number of apps**: More apps means more diversity over tools. When generating apps, we ensure an equal balance across the 8 taxonomy categories previously defined.
- T **number of tasks per-app**: This is the number of tasks generated based on functionalities within a particular app. A higher T means more diversity of tasks within an app.
- S **number of initial databases per-task instruction**. We can keep the same task instruction, verifier, and oracle solver but vary the initial database state Db^0 to form different tasks.
- D **amount of distractor data** per initial database state. This is the total number of data batches, including the core batch $\text{Db}^0_{\text{core}}$, that make up the initial database state.

The task set diversity is characterized by A, T, S, producing $N = A \cdot T \cdot S$ total tasks. The data complexity is characterized by D, which does not change the number of tasks.

4 Experiments

4.1 Experimental Setup

We now conduct an analysis to determine the relationship between the SynthAPI controls described in Section 3.2 and how agents trained on the generated data generalize to downstream agent benchmarks. We thus use SynthAPI to generate many task sets $\{(\mathcal{A}_i, \mathcal{T}_i)\}$ based on different sets of apps and then train an agent on each task set independently. In total we generate 458,752 tasks, across 64 apps which consist of 10,472 APIs with 6,266 write APIs and 4,206 read APIs. The median number of APIs per app is 144 but it ranges per app with a minimum of 51 and a maximum of 443. The generated APIs take between 0 and 12 input arguments. The median number of data class definitions per database schema implementation is 55 and it ranges from 10 to 135. Apps mostly have between 31 and 143 functionalities with a median of 81. The median number of functionalities per generated task is 2 ranging from 1 to 7. The core database state $\text{Db}^0_{\text{core}}$ has a median of 59 rows. See Section A.4 for more statistics and full histograms about the generated tasks and apps.

Each episode includes a binary verifier, so agents can be directly trained in SynthAPI via reinforcement learning with verifiable rewards. However, running an RL loop for every setting in our analysis is prohibitively expensive, so we instead collect trajectories once and then use supervised fine-tuning (SFT) to train on subsets $\mathcal{T}_i$ at the given control settings. We use GLM-5.2 (GLM-5 Team, 2026) as the expert and allow it up to four attempts per episode. We retain the first attempt for which the verifier indicates successful task completion and discard episodes for which none of the four attempts succeeds. The task counts used throughout this analysis thus each have exactly one trajectory. We

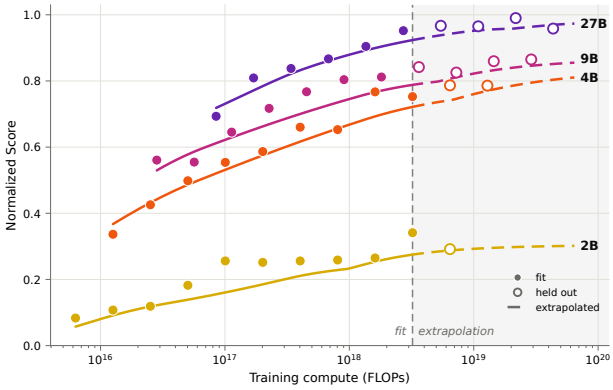


Figure 3: Visualization of the proposed data composition dependent scaling law and its extrapolation to higher compute budgets. The x-axis reports the training FLOPs of the SFT training and the y-axis is the normalized score indicating the combined generalization performance across various benchmarks. The points are normalized scores for models trained on an optimal setting, meaning the data setting that performs the best at that FLOP budget, representing the FLOP/performance allocation frontier. The lines are a visualization of the per parameter count scaling model, fit without the held-out runs in the top 25% of training FLOPs.

distill the retained expert trajectories with the standard next-token prediction objective into student models from the Qwen3.5 family (Team, 2026). The loss is applied only to the reasoning and action tokens. By default, we train models for 1 epoch over the dataset. Training continues until the validation loss increases over a set number of updates, after which we select the checkpoint with the lowest validation loss for evaluation. This dataset constitutes 5.9B total tokens of which 721M are agent reasoning and action output for actual supervision. We train 849 models across the different task populations controlling for the different dataset composition properties. Section B lists all training details and more statistics about the trajectory dataset.

We evaluate on a representative set of benchmarks containing multi-step tool use in tools and tasks that are distinct from the SynthAPI environments we train on, thus evaluating generalization. We focus on benchmarks where the user in the loop is not a primary challenge as SynthAPI does not generate data for these scenarios. All tasks are evaluated based on the success rate of the agent being able to achieve the stated goal while avoiding unintended changes.

- **AppWorld** (Trivedi et al., 2024) evaluates agents in device assistant tasks across 9 everyday apps and 457 APIs. Tasks involve complex user data and combining information across apps in the face of distractors and constraints. We evaluate on the Test Normal (Test-N) of 168 tasks and Test Challenge (Test-C) of 417 tasks.
- **BFCLv3 Multi-Turn** (Patil et al., 2025) has 129 APIs across 8 domains. We restrict to the multi-turn split which has 800 tasks which also explicitly test robustness to missing APIs, missing parameters, and long tool outputs.
- **ASTRA-Bench** (Xiu et al., 2026) focuses on persona-driven evaluation, where each task combines rich user context with 27 APIs across 6 apps. We evaluate on the most complex "High" setting consisting of 939 tasks.
- **SynthAPI-Holdout.** We construct a held-out set of new apps, tasks and data profiles using SynthAPI allowing us to measure within-distribution cross-app generalization. We generate 4 apps with 25 tasks per app where each task is generated with the maximum data complexity setting.

We use a subset of these benchmarks, which we call the *diagnostic set*, for the scaling and data composition analysis. AppWorld and SynthAPI-Holdout are used as the diagnostic set because evaluating on these benchmarks alone required 1.8 million episodes. Because raw scores are not directly comparable across benchmarks, we normalize the model's score when doing analysis on the diagnostic set. To do so, for benchmark i and model size M, let θ_0^M be the starting model parameters, Θ_i the population of all trained models across all sizes and $\text{Succ}_i(\theta)$ the success rate of model parameterized by θ averaged over 4 trials per episode. We then compute normalized score as $P(\theta^M) = \frac{1}{K}\sum_{i=1}^{K} \frac{\text{Succ}_i(\theta^M) - \text{Succ}_i(\theta_0^M)}{\max_{\theta' \in \Theta_i} \text{Succ}_i(\theta') - \text{Succ}_i(\theta_0^M)}$. P is thus bounded between $[0, 1]$ starting at the zero-shot performance of the Qwen student without SynthAPI training and ending at the maximum

	AppWorld			ASTRA-Bench			BFCL			SynthAPI-Holdout		
Approach	4B	9B	27B	4B	9B	27B	4B	9B	27B	4B	9B	27B
Zero-shot	14.7	26.4	61.5	57.4	71.8	80.2	27.3	39.3	54.4	0.3	1.5	20.0
Nemotron	17.5	20.8	-	54.4	67.7	-	15.9	39.7	-	1.0	0.5	-
SynthAPI-SubOpt	38.4	53.0	73.8	71.1	74.2	77.3	44.9	47.5	59.6	51.3	56.0	64.0
SynthAPI-Opt	45.0	56.2	74.1	72.3	75.5	79.3	48.8	49.9	61.5	**56.0**	63.0	68.0
SynthAPI-Big	**66.4**	**77.0**	**81.5**	**74.2**	**76.8**	**82.3**	**51.1**	**53.6**	**65.0**	53.0	**64.0**	**69.8**

Table 1: Unnormalized success rates on downstream tool-agent benchmarks. AppWorld reports the Test-C split (see Table 4 for both splits). SynthAPI rows are trained with SynthAPI data; "-Opt" uses the optimal data composition from our scaling analysis, and "-Big" is a scaled-up version.

observed training performance on this benchmark. Section D contains complete benchmark details and a full description of the evaluation protocol.

4.2 Task Properties Analysis

We first vary the individual SynthAPI task-generation controls described in Section 3.2 under fixed task or data budgets and evaluate downstream generalization on the diagnostic set.

Tasks vs. apps: First, we study how to allocate a fixed task budget between generating more apps, and thus more APIs, and generating more tasks for existing apps. The total task count is $N = AT$, where A is the number of apps and T is the number of tasks per app. For this comparison, we fix the number of databases per task at $S = 1$ and the distractor-data setting at $D = 2$. We sweep all combinations of $A \in \{8, 16, \ldots, 64\}$ and $T \in \{8, 12, 16, \ldots, 1024\}$, training a separate model on the tasks generated for each configuration. We evaluate each trained model π_θ on the diagnostic benchmarks to obtain $P(\theta)$. For each app count A, we model the relationship between P and N as T varies. We fit the following generalized logistic model to the observed data using least-squares minimization, estimating parameters U, b, b_N, ν separately for each model size M and app count A.

$$\hat{P}_{M,A}(N) = U \cdot \sigma\left(b + b_N \log_2 N\right)^\nu \tag{1}$$

Figure 2a compares allocations of A and T and their fitted curves across training task budgets N for the 9B model. At lower task budgets, fewer apps yield better performance. We hypothesize that concentrating tasks within a smaller set of tools and data structures yields larger initial gains. As the task budget increases, however, a larger number of apps becomes preferable. Specifically, $A = 8$ performs best at 64–512 tasks, $A = 16$ at 512–1024, and $A = 64$ at 1024–4096. The fitted performance ceiling also increases with A, with $A = 8$ saturating at a substantially lower value than $A = 64$. The gains from increasing app diversity are not due to tasks exhausting each app's functionality. Even at $T = 1024$ tasks per app, the median fraction of tools used is only 51%, reflecting the complexity of SynthAPI apps in their tool counts and input-argument counts and types (Section A.4). Prior work such as Tu et al. (2026) also finds benefits from scaling the number of tools, but increases training domains and data together rather than comparing allocations under a fixed task budget. Section E.1 shows the same trends across the 2B, 4B, and 27B model sizes.

Data profiles vs. tasks: We then study database breadth by varying S, the number of database instantiations generated per task instruction. Each instantiation preserves the instruction but changes the concrete entities, values, and relations the agent encounters. The x-axis shows the total number of training instances N, and each curve corresponds to a different value of S. For each budget N, we use the optimal app-to-task allocation from the previous study, adjusted for the selected S so that $N = T \cdot A \cdot S$. Each task instruction receives its own distinct data profiles. We fit the model in Equation (1) separately for each database count S in place of app count A, yielding $\hat{P}_{M,S}$. The results in Figure 2b show that higher task budgets favor greater database breadth. At budgets of 32–512 training instances, one data profile per instruction performs best; at 512–4096, two profiles perform best; and beyond 4096, four profiles perform best. The fitted performance ceiling also increases with S. We attribute these gains to learning to solve the same task across different initial database states, since data manipulation is a key challenge in our training tasks. This suggests that, as the budget grows, generating additional data profiles for existing instructions becomes preferable to adding new instructions with new data profiles.

BFCL	4B	8B
Zero-shot	33.5	41.3
Agent World Model	40.8	42.3
EnvScaler	45.0	51.9
EnvFactory	48.5	49.0
SynthAPI-Big	**50.5**	**55.6**

	4B		9B	
AppWorld	Test-N	Test-C	Test-N	Test-C
Zero-shot	47.5	14.7	45.4	26.4
ToolACE	20.2	10.9	18.6	12.4
ToolAlpaca	25.7	17.3	24.4	15.7
SimSAT	66.8	58.8	74.5	65.4
SynthAPI-Big	**81.0**	**66.4**	**83.3**	**77.0**

Table 2: Comparison to per-benchmark baselines from prior work on BFCL (left) and AppWorld (right) showing success rates. BFCL results are on Qwen3 and AppWorld results are on Qwen3.5.

Effect of data complexity: We also study database depth, which controls the amount of distractor data within each data profile. We compare $D = 1$, which includes only the minimum state needed to solve the task; $D = 2$, which adds a moderate amount of distractor data; and $D = 4$, which adds a larger distractor set. As before, the x-axis shows the task budget N, the y-axis shows aggregate downstream performance, and each curve corresponds to a different D; we use the optimal allocation of A and T with $S = 1$. Figure 2c shows a large improvement from $D = 1, 2$ to $D = 4$ at high task budgets, indicating that training with distractor data improves downstream scaling. This pattern may partly reflect the current downstream benchmarks' limited sensitivity to increasingly difficult database states. Benchmarks that place greater emphasis on retrieval and decision-making over dense database states may exhibit different scaling behavior along this dimension.

4.3 Scaling Law

In this section, we take the learned lessons from the previous section to now fit a predictable relationship between all of the SynthAPI controls and the diagnostic set performance. In total, we use all 849 trained models across all the different dataset populations which pairs each setting of (A, T, S, D) with downstream normalized score across each model size M. Guided by the previous analysis, we formulate the scaling law equation to build off of Equation (1) but now fit a data composition dependent upper limit $U(A, S, D)$ rather than the previous fixed U value. Specifically, we use the following form, where c_0, c_A, b_0, b_T, ν are fit from the data using least-squares minimization.

$$\widehat{P}^1_M(A, T, S, D) = \sigma(c_0 + c_A \log_2(A \cdot S \cdot D))\sigma(b_0 + b_T \log_2 T)^\nu \tag{2}$$

We also compare to a form that only uses the total number of trajectories $\hat{P}^0_M(N) = U\sigma(b_0 + b_N \log_2 N)^\nu$. We measure the FLOPs of each training run with the standard Chinchilla approximation $C \approx 6ML$ (Hoffmann et al., 2022) where L is the number of total supervision tokens in the training trajectories. We hold out the runs in the top 25% of training FLOPs for extrapolation and fit only on the remaining runs, thus evaluating a 16x extrapolation increase in FLOP budget.

Figure 3 visualizes the data composition dependent scaling curve across each model size along with the actual observed frontier optimal data points at various FLOP budgets. We only display the optimal points at the given FLOP budgets, but the curves are fit to all the data points. The data composition aware scaling law achieves a held-out mean absolute error (MAE) value of 0.0285 in the extrapolation region (the top 25% of training FLOPs) compared with an MAE of 0.1706 for the data-quantity-only scaling law $\hat{P}^0_M$. Section E.1 reports the fit quality for each model size. The findings are also the same if $\hat{P}^0_M$ is parameterized by the token count, and equivalently the FLOPs, instead of the number of tasks N. This demonstrates that the proper composition of the dataset is important for agent scaling rather than just sheer scale. The data dependent scaling law shows that the upper ceiling of performance is increased by increasing the app count, number of data profiles per task and data profile complexity with the number of tasks per app controlling the rate of progress towards this maximum. Furthermore, we demonstrate it is possible to extrapolate to predict agent performance under up to $16\times$ larger compute budgets.

4.4 Baseline Comparison

Next, we demonstrate the performance of SynthAPI across all the downstream benchmarks and compare to prior works in task synthesis. Now we also report performance on the ASTRA-Bench and BFCLv3 benchmarks. First, to show the value of the FLOP optimal training data composition

settings derived from the previous scaling law analysis we compare to two versions of models trained with SynthAPI data and a large scale existing tool-agent dataset:

- SynthAPI-SubOpt: We train one model on 32,768 SynthAPI tasks, but we ignore the lessons from the previous sections and instead choose $A = 32$, $T = 1024$, $S = 1$, $D = 2$.
- SynthAPI-Opt: We train under the same task budget as the above, but now maximize app and database diversity by reducing task prompts, with $A = 64$, $T = 128$, $S = 4$, $D = 4$.
- Nemotron (NVIDIA, 2025): We train on the full Nemotron-Agentic dataset.
- SynthAPI-Big: We train on the full SynthAPI generated dataset but now for 4 epochs.

The results from Table 1 show the performance of these approaches across all benchmarks. We omit the 2B size due to it performing far worse than the other sizes. First comparing SynthAPI-Opt and SynthAPI-SubOpt, we see the value of the right data recipe with increasing S, D and A while keeping the total task budget fixed. SynthAPI-Opt does consistently better than SynthAPI-SubOpt, with especially large deltas in SynthAPI-Holdout and the 4B model. When scaling up to SynthAPI-Big, results further improve on most benchmarks with especially large gains in AppWorld. This method greatly outperforms Nemotron, which contains relatively simple and short-horizon data that does not generalize to the complexity of these benchmarks and even underperforms the zero-shot starting model in 5 of the 8 settings.

Next, we compare with results reported in prior work in Table 2. On AppWorld, we compare with **ToolACE** (Liu et al., 2024a) and **ToolAlpaca** (Tang et al., 2023), which synthesize tool-use demonstrations with LLM-simulated API responses, and **SimSAT** (Lee et al., 2026), which uses LLM simulation to synthesize long-horizon trajectories. On BFCL, we compare with **Agent World Model** (Wang et al., 2026), which synthesizes database-backed tool environments; **EnvScaler** (Song et al., 2026), which generates environment skeletons and verifiable task scenarios; and **EnvFactory** (Xu et al., 2026), which builds executable environments from real-world resources. Since these BFCL baselines only report results for Qwen3, we also train SynthAPI-Big on Qwen3-4B/8B.

ToolACE and ToolAlpaca generally underperform the zero-shot models on AppWorld, indicating limited transfer to its complex stateful tasks. SimSAT, which uses LLM simulation of long-horizon interactions and GLM-based distillation, provides a substantially stronger baseline. SynthAPI-Big achieves the highest scores on both AppWorld splits, reaching 66.4% and 77.0% on Test-C at 4B and 9B, respectively, exceeding SimSAT by 7.6 and 11.6 percentage points. On BFCL, the prior executable environment synthesis methods all improve over zero-shot, while SynthAPI-Big further surpasses the strongest baseline by 2.0 percentage points at 4B (EnvFactory) and 3.7 points at 8B (EnvScaler). However, their training regimes differ substantially with these methods combining SFT and RL on smaller datasets, whereas SynthAPI-Big uses SFT on a much larger synthetic corpus. These comparisons therefore do not isolate the effect of the synthesis method or establish relative data efficiency. They do demonstrate that SynthAPI supports large-scale synthesis of tasks that enable strong downstream generalization. The strong results overall show how our scaling laws can guide the construction of effective training datasets. See Section E for more results on transferring findings to other datasets, evaluation breakdowns, and further fit quality analysis.

5 Conclusion

This work introduces SynthAPI, a controllable pipeline for synthesizing executable environments for training and studying tool agents. SynthAPI uses LLMs to generate executable environments with a structured database state and Python API implementations. SynthAPI then generates verifiable tasks with a starting database state. This design allows us to independently scale the diversity of tools, data profiles, and tasks. We empirically show that the composition of training data substantially affects downstream performance. We combine these results in a joint scaling law that predicts downstream performance and gives optimal data composition settings for a given compute budget. A limitation of SynthAPI is that it does not simulate a user in the loop that interacts with the agent. Such a setting is important for realistic settings where agents need to query a user to ask for additional details or clarifying questions. Furthermore, SynthAPI focuses less on information retrieval tasks involving long and complex documents in realistic user settings like in emails, product reviews or notes. Since all content in SynthAPI is synthesized from scratch by the LLM it lacks this semantic depth. Furthermore, the SynthAPI scaling law is only validated on a single data pipeline and it is not guaranteed to transfer to different data pipelines or training settings.

AI USE STATEMENT

In this work, we used generative AI tools for implementing methods, generating synthetic data sets, and cleaning and reformatting datasets. We have not used generative AI tools for proposing or refining hypotheses, designing or providing feedback on research methodology or experiments, assisting with translation, supporting qualitative and thematic data analysis, or interpreting results. Helping develop theoretical models or conceptual frameworks, formulating mathematical claims, providing critical ingredients for proving mathematical claims, and assisting in the writing of proofs are not applicable to this work. Additionally, we used generative AI tools for creating (scripts that create) scientific figures.

The usage of coding agents in this paper should be divided into two categories: 1) The purpose of this work was to build an automated dataset generation workflow at scale, hence the goal was to have the final builder LLM be as autonomous as possible, and obtain an automatically generated train dataset. Throughout the development, we controlled and refined the flow of how the builder LLM builds environments, and manually checked rollouts generated by policies trained on SynthAPI to detect potential reward hacking, but otherwise left the exact implementations of apps, APIs, tasks, and databases relatively unsupervised. The proof of the generated train data is in the evaluation performance, after all. 2) The evaluation, on the other hand, was strictly controlled. The evaluation datasets AppWorld, BFCL, and ASTRA-Bench are human-created, and our implementations around them are largely human-generated as well. Whenever coding agents were used, they underwent the same PR review process as any other change, where two of our authors review and test against unit tests.

We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Aili Chen, Chi Zhang, Junteng Liu, Jiangjie Chen, Chengyu Du, Yunji Li, Ming Zhong, Qin Wang, Zhengmao Zhu, Jiayuan Song, et al. Dive: Scaling diversity in agentic task synthesis for generalizable tool use. *arXiv preprint arXiv:2603.11076*, 2026.

Zehui Chen, Kuikun Liu, Qiuchen Wang, Wenwei Zhang, Jiangning Liu, Dahua Lin, Kai Chen, and Feng Zhao. Agent-FLAN: Designing data and methods of effective agent tuning for large language models. *arXiv preprint arXiv:2403.12881*, 2024. URL https://arxiv.org/abs/2403.12881.

Guanting Dong, Junting Lu, Junjie Huang, Wanjun Zhong, Longxiang Liu, Shijue Huang, Zhenyu Li, Yang Zhao, Xiaoshuai Song, Xiaoxi Li, et al. Agent-world: Scaling real-world environment synthesis for evolving general agent intelligence. *arXiv preprint arXiv:2604.18292*, 2026.

GLM-5 Team. GLM-5: From vibe coding to agentic engineering. *arXiv preprint arXiv:2602.15763*, 2026. URL https://arxiv.org/abs/2602.15763.

Google DeepMind. Gemini 3.1 pro: Model card. https://deepmind.google/models/model-cards/gemini-3-1-pro/, February 2026. Published February 19, 2026.

Danny Hernandez, Jared Kaplan, Tom Henighan, and Sam McCandlish. Scaling laws for transfer. *arXiv preprint arXiv:2102.01293*, 2021. URL https://arxiv.org/abs/2102.01293.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022. URL https://arxiv.org/abs/2203.15556.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020. URL https://arxiv.org/abs/2001.08361.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Seanie Lee, Sanjoy Chowdhury, Chao Jiang, Cheng-Yu Hsieh, Ting-Yao Hu, Alexander T Toshev, Oncel Tuzel, and Raviteja Vemulapalli. Environment-free synthetic data generation for api-calling agents. *arXiv preprint arXiv:2607.16900*, 2026.

Weiwen Liu, Xu Huang, Xingshan Zeng, Xinlong Hao, Shuai Yu, Dexun Li, et al. Toolace: Winning the points of llm function calling. *arXiv preprint arXiv:2409.00920*, 2024a.

Zuxin Liu et al. APIGen: Automated pipeline for generating verifiable and diverse function-calling datasets. *arXiv preprint arXiv:2406.18518*, 2024b. URL https://arxiv.org/abs/2406.18518.

NVIDIA. Nemotron-Agentic-v1. Hugging Face dataset, 2025. URL https://huggingface.co/datasets/nvidia/Nemotron-Agentic-v1.

Shishir G. Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 48371–48392. PMLR, 2025.

Akshara Prabhakar et al. APIGen-MT: Agentic pipeline for multi-turn data generation via simulated agent-human interplay. *arXiv preprint arXiv:2504.03601*, 2025. URL https://arxiv.org/abs/2504.03601.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. *arXiv preprint arXiv:2307.16789*, 2023. URL https://arxiv.org/abs/2307.16789.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *Advances in Neural Information Processing Systems*, 2023. URL https://arxiv.org/abs/2302.04761.

Xiaoshuai Song, Haofei Chang, Guanting Dong, Yutao Zhu, Zhicheng Dou, and Ji-Rong Wen. EnvScaler: Scaling tool-interactive environments for LLM agent via programmatic synthesis. *arXiv preprint arXiv:2601.05808*, 2026. URL https://arxiv.org/abs/2601.05808.

Zelin Tan et al. Scaling behaviors of LLM reinforcement learning post-training: An empirical study in mathematical reasoning. *arXiv preprint arXiv:2509.25300*, 2025. URL https://arxiv.org/abs/2509.25300.

Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. Toolalpaca: Generalized tool learning for language models with 3000 simulated cases. *arXiv preprint arXiv:2306.05301*, 2023.

Qwen Team. Qwen3. 5-omni technical report. *arXiv preprint arXiv:2604.15804*, 2026.

Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, et al. Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*, 2024.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. *arXiv preprint arXiv:2407.18901*, 2024. URL https://arxiv.org/abs/2407.18901.

Dunwei Tu, Hongyan Hao, Hansi Yang, Yihao Chen, Yi-Kai Zhang, Zhikang Xia, Yu Yang, Yueqing Sun, Xingchen Liu, Furao Shen, Qi Gu, Hui Su, and Xunliang Cai. ScaleEnv: Scaling environment synthesis from scratch for generalist interactive tool-use agent training. *arXiv preprint arXiv:2602.06820*, 2026. URL https://arxiv.org/abs/2602.06820.

Zhaoyang Wang, Canwen Xu, Boyi Liu, Yite Wang, Siwei Han, Zhewei Yao, Huaxiu Yao, and Yuxiong He. Agent world model: Infinity synthetic environments for agentic reinforcement learning. *arXiv preprint arXiv:2602.10090*, 2026. URL https://arxiv.org/abs/2602.10090.

Zidi Xiu, David Q. Sun, Kevin Cheng, Maitrik Patel, Josh Date, Yizhe Zhang, Jiarui Lu, Omar Attia, Raviteja Vemulapalli, Oncel Tuzel, Meng Cao, and Samy Bengio. ASTRA-bench: Evaluating tool-use agent reasoning and action planning with personal user context. *arXiv preprint arXiv:2603.01357*, 2026. URL https://arxiv.org/abs/2603.01357.

Minrui Xu, Zilin Wang, Mengyi Deng, Zhiwei Li, Zhicheng Yang, Xiao Zhu, Yinhong Liu, Boyu Zhu, Baiyu Huang, Chao Chen, et al. Envfactory: Scaling tool-use agents via executable environments synthesis and robust rl. *arXiv preprint arXiv:2605.18703*, 2026.

Ted Zadouri, Markus Hoehnerbach, Jay Shah, Timmy Liu, Vijay Thakkar, and Tri Dao. Flashattention-4: Algorithm and kernel pipelining co-design for asymmetric hardware scaling, 2026.

Jianguo Zhang, Tian Lan, Ming Zhu, Zuxin Liu, Thai Hoang, Shirley Kokane, Weiran Yao, Juntao Tan, Zhiwei Liu, Yihao Feng, et al. xlam: A family of large action models to empower ai agent systems. 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)*, pp. 11583–11597, 2025.

Yuchen Zhuang, Jingfeng Yang, Haoming Jiang, Xin Liu, Kewei Cheng, Sanket Lokegaonkar, Yifan Gao, Qing Ping, Tianyi Liu, Binxuan Huang, et al. Hephaestus: Improving fundamental agent capabilities of large language models through continual pre-training. 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)*, pp. 6041–6068, 2025.

A SynthAPI—Full Details

A.1 App Generation Details

Functionality Generation Prompt. Sent to the builder LLM to generate the functionalities of a given app, which are later sampled to construct task instructions.

```
Give me every possible thing I can do in the {app} app, be
    comprehensive. Do not include any functionality that requires
    interacting with another app (for example, importing data from
    another app). Functionality that can be done entirely in-app is
    fine (for example, managing connections within {app} itself).
    Focus on the core, domain-specific things a user does in {app} (
    for example, the products, content, or services the app is
    actually about). Do not produce functionality that is solely
    about account management (for example, signing up, logging in,
    changing your password, updating email or notification
    preferences, or managing account settings).
```

Database Schema Generation Prompt. Used to generate an app's database schema (`db.py`) as a collection of Python data classes, given the app metadata: the app name, its description, and the functionality list produced by the preceding prompt.

```
I want to create a data model of objects in a relational database.
    Specifically, I want to mock up a version of the {app} app with
    data models that can back any possible functionality in the app.
     I want this expressed as a series of Python dataclass objects.
    I want an exhaustive description of all possible data I would
    need for every possible interaction in the {app} app. Write all
    of the dataclasses into db.py and do not stop until every
    possible data type is covered. Make sure the data classes cover
    all of the functionalities listed below.

Functionalities:
```
{functionalities}
```
```

The generation here uses the builder LLM, which we use Gemini-3.1 Pro for. In the app creation, we use Gemini-3.1 Pro through the Gemini CLI harness. This allows the model to agentically create the various app components and take on more complex changes. The Gemini CLI harness is only used for the app creation and not for the task generation.

API Generation Prompt. Used to generate an app's API implementation as ordinary Python functions, conditioned on the previously generated database schema.

```
I want to mock up all of the APIs in {app}. Implement all of the
    APIs in apis.py. To build out the APIs you are only allowed to
    use the four CRUD functions the database exposes, and you must
    build APIs using all of the data models in db.py. Your APIs must
     cover any possible task across the functionalities listed below
    . Decorate every function that does not require accessing the
    current user (for example searches and public catalog lookups)
    with the `no_auth` decorator. Functions that need the current
    user must not use this decorator.

Data models:
```
{db_models}
```
```

Functionalities:
```
{functionalities}
```
```

When implementing the APIs, we only allow an API to interact with the database state via one of four defined create, read, update, delete functions. We use this access gating so we can directly track all changes made to the database and keep this access log in the database object.

### A.2 Task Generation Details

**Task Generation Prompt.** Used to generate unambiguous task instructions from a sampled subset of an app's functionalities.

```
You are generating task instructions for an AI agent that interacts
     with the {app} app via APIs.

Below are functionalities of the app:
```
```

```
{functionalities}

Existing tasks (for reference, do NOT duplicate these):
{existing_tasks}

Generate exactly {n_tasks} diverse, concrete task instructions that
     exercise the functionalities above. Each task should:
1. Be SHORT: a single, clear instruction phrased the way a real
    person casually speaks it, in 1-2 sentences (roughly 15-40 words
    ), NOT a long multi-step plan.
2. Explicitly mention the app name "{app}" in the task instruction
3. Include specific product names, quantities, or criteria where
    appropriate
4. Vary in difficulty (some simple, some multi-step)
5. Not duplicate any existing task
6. Be an actionable instruction (e.g. "add X to my cart", "place an
     order for Y"), NOT a question that requires an answer (e.g. "
    what is the price of X?", "how many Y are in stock?"). The agent
     should be performing actions, not retrieving information to
    answer the user.{additional_rules}

Keep tasks SHORT - the difficulty should come from a single
    constraint or decision the agent must reason about, not from
    stacking many steps.

Output your tasks as a Python list of strings:
'''python
tasks = [
   "...",
   "...",
]
'''
```

Initial State Generation Prompt. Used to instantiate the core initial database state containing the records and relations necessary to make a task instruction meaningful and solvable.

```
Using the defined data models, write a setup_database function that
     returns a list of dataclass instances to populate the database
    with initial data. The data should:
1. Include all entities needed so that the task instruction below
    can be successfully completed
2. Include some distractor data (extra realistic entries that aren'
    t directly needed for the task) to make the scenario more
    realistic
3. Use consistent IDs and foreign key references between related
    entities
4. Use realistic values (names, emails, prices, etc.)
5. Any user-related data (User, Address, etc.) for the main user
    MUST use the exact values shown in the "Main user" section below
    .

The output should be in format:
'''python
def setup_database() -> list:
   ...
'''
Do not output any other code. Do not include any import statements
    and reference the data models exactly as they are named.

Data models:
```

```
```
{db_models}
```

APIs (for reference on available operations):
```
{api_docs}
```
```

The actual database state is thus just a list of instantiated dataclass objects. We also couple the database with a change log of executed create, update, read or delete operations that happened on the database.

Verifier Generation Prompt. Used to generate the programmatic verifier that determines whether a database state satisfies the task instruction.

```
Using the defined APIs, write a verifier for the below "Instruction
    " to check if the task is successfully completed. The output
    should be in format:
```python
def verify(db: Database, answer: str | None = None) -> bool:
    ...
```
Do not output any other code. Do not include any import statements
    and reference the APIs exactly as they are named.

APIs:
```
{api_docs}
```
Instruction: `{instruction}`
```

Oracle Generation Prompt. Used to generate the oracle solver program that completes the task instruction by calling app APIs from the initial database state.

```
Using the defined APIs, write a solve function that performs the
    actions described in the instruction to complete the task. The
    solve function receives a Database object that has been pre-
    populated with initial data.

Hard requirements:
- Only call the public APIs listed below to make changes. Do not
    call, or read attributes from, anything whose name starts with `
    _` (private/internal helpers are off-limits).
- Do not call `create_data_model`, `read_data_model`, `
    update_data_model`, or `delete_data_model` directly, do not use
    `setattr`, and do not read or write `db` (e.g. `db.data`, `db.<
    field> = ...`, `db.<field>.append(...)`) directly. Every change
    to the database must go through a public API call.

The output should be in format:
```python
def solve(db: Database) -> None:
    ...
```
Do not output any other code. Do not include any import statements
    and reference the APIs exactly as they are named.
```

```
Data models:
```
{db_models}
```

APIs:
```
{api_docs}
```

solve_task definition (for context on how solve is called):
```
{solve_task_def}
```
Instruction: `{instruction}`

Task file (for context on the setup data and verifier):
```
{task_file}
```
```

Database Expansion Prompt. Used to expand the core initial database state with distractor records that increase retrieval and selection difficulty without changing the intended solution.

```
Using the defined data models, write a setup_additional_stage_{
    stage_n} function that returns a list of dataclass instances to
    add EXTRA distractor data on top of the data already produced by
     setup_database AND by the earlier distractor stages (
    setup_additional_stage_1 .. setup_additional_stage_{prev_n}).
    The goal is to make the task harder by introducing realistic but
     irrelevant entries (e.g. additional users, similar-looking
    orders, products in the same category, etc.).

This is stage {stage_n} in a sequence of distractor stages. When
    the task runs, the records returned by setup_database and by
    EVERY setup_additional stage (1 through {stage_n}) are all
    loaded into the SAME database together. Your stage {stage_n}
    data must therefore coexist with everything the earlier stages
    already added. To keep the overall distractor set diverse, your
    stage {stage_n} data SHOULD be meaningfully DIFFERENT from what
    the earlier stages produced (shown in the "Earlier
    setup_additional stages" section below) - introduce new kinds of
     realistic-but-irrelevant records rather than repeating the same
     patterns, entities, or IDs.

Hard requirements:
1. The additional data MUST NOT cause the verifier to return True
    before the solve runs (the task must still start unsolved).
2. The additional data MUST NOT prevent the solve function from
    completing successfully (the task must still be solvable).
3. Use IDs that do not collide with anything already created in
    setup_database OR in ANY earlier setup_additional stage.
4. The additional data SHOULD be relevant to the main user (shown
    in the "Main user" section below) - e.g. extra orders, addresses
    , payment methods, reviews, browsing history, or other records
    belonging to or associated with the main user - so the
    distractors are plausible noise in the main user's account.
    These records must NOT interfere with the solvability of the
    main task or cause the verifier to pass prematurely.
```

```
The output should be in format:
```python
def setup_additional_stage_{stage_n}() -> list:
    ...
```
Do not output any other code. Do not include any import statements
    and reference the data models exactly as they are named.

Data models:
```
{db_models}
```

APIs (for reference on available operations):
```
{api_docs}
```
```

A.3 Agent Interface Details

Each task is a Gymnasium environment (Towers et al., 2024) whose state is the instantiated database and whose transitions are app API calls. The environment is deterministic, so a trajectory is fully determined by Db^0 and the agent's actions.

We follow an action format similar to the AppWorld environment (Trivedi et al., 2024): an action is a single fenced Python code block. It executes in a Python session that persists for the whole episode, so variables defined on one turn are available on later turns. The app API functions and the episode's `Database` object are bound in the session namespace, so the agent calls tools as ordinary Python functions. A single turn can therefore call several tools, loop over the records returned by an earlier call, and branch on intermediate values. Text outside the code block is treated as reasoning and is not executed. If a turn contains no parseable code block, the environment returns a format error and the turn counts against the step limit.

The observation is the output of the previous code block execution, with tool results returned in JSON format. When return values are `db.py` dataclasses, or lists of them, we serialize them field by field into JSON so that field names and values are explicitly displayed to the agent rather than generic class identifiers. In other words, when an API returns an instantiated dataclass, the agent sees its full data contents. Errors and exceptions do not end the episode. Instead, the error message is displayed and the agent can fix the problem on the next turn. Observations are truncated to 2,048 tokens with an explicit `[truncated]` marker.

We also expose several "meta-tools" which are always present in the agent's toolbox. The meta-tools are used to discover and look up documentation for the dynamic per episode tool set the agent has to deal with.

- `list_apps()`: name and one-line description of each app in the episode.
- `list_tools(app)`: names and one-line summaries of one app's public API functions.
- `get_tool_documentation(app, tool)`: one tool's description, argument names, types and defaults, and return type.
- `complete_task(answer=None)`: ends the episode, with an optional answer for tasks that require reporting information rather than only changing state.

Episodes in SynthAPI are capped at 32 steps. The verifier is called only when the agent invokes `complete_task`, so a task counts as a success only if the agent calls `complete_task` and v then returns true on the resulting database state. An agent that reaches the step limit without calling `complete_task` fails the task, even if its actions happened to satisfy the verifier.

Below is the initial task prompt given to the agent as the first user message. Information in the templates is dynamically substituted based on the starting database state and task instruction.

````
I am your supervisor and you are a super intelligent AI Assistant
    whose job is to achieve my day-to-day tasks completely
    autonomously.

To do this, you will need to interact with apps using their
    associated APIs on my behalf. For this you will undertake a *
    multi-step conversation* using a python REPL environment. That
    is, you will write the python code and the environment will
    execute it and show you the result, based on which, you will
    write python code for the next step and so on, until you've
    achieved the goal. This environment will let you interact with
    app/s using their associated APIs on my behalf.

Here are three key tools that you need to know to get more
    information

# To get a list of apps that are available to you.

```python
print(list_apps())
```

# To get the list of tools under any app listed above

```python
print(list_tools(app='xyz'))
```

# To get the specification of a particular tool

```python
print(get_tool_documentation(app='xyz', tool='abc'))
```

**Key instructions**:
- Remember you can use the variables in your code in subsequent
    code blocks.
- ALWAYS look at tool specifications (using get_tool_documentation)
     before calling a tool. Never call a tool without first using
    get_tool_documentation.
- Write small chunks of code and only one chunk of code in every
    step. Make sure everything is working correctly before making
    any irreversible change.
- Use `for` loops when processing lists of data rather than
    manually handling items one by one.
- Check the environment state before proceeding. The task may start
     in the middle of a user session, meaning the user may have
    already modified the environment. Do not assume the environment
    is in a fresh or default state.
- Use `print` to inspect code outputs.
- Once you have completed the task, make sure to call complete_task
    (). If the task asked for some information, return it as the
    answer argument, i.e. call complete_task(answer=<answer>). Many
    tasks do not require an answer, so in those cases, just call
    complete_task() i.e. do not pass any argument.
- Output just a single Python code block.

Using these APIs, now generate code to solve the actual task:

My name is: {{ main_user.first_name }} {{ main_user.last_name }}.
    My personal email is {{ main_user.email }} and phone number is
````



(a) APIs / App (b) Data Classes / App (c) Functionalities / App

(d) Input Arguments / API (e) Required IDs / API (f) Parameter Types

Figure 4: Statistics of the generated apps and their APIs. The top row shows per-app statistics and the bottom row shows per-API and per-parameter statistics. The sample used here is the complete set of 64 apps used in all the experiments.

```
    {{ main_user.phone_number }}. The current date is {{ main_user.
    session_start_time }}.
Task: {{ input_str }}
```

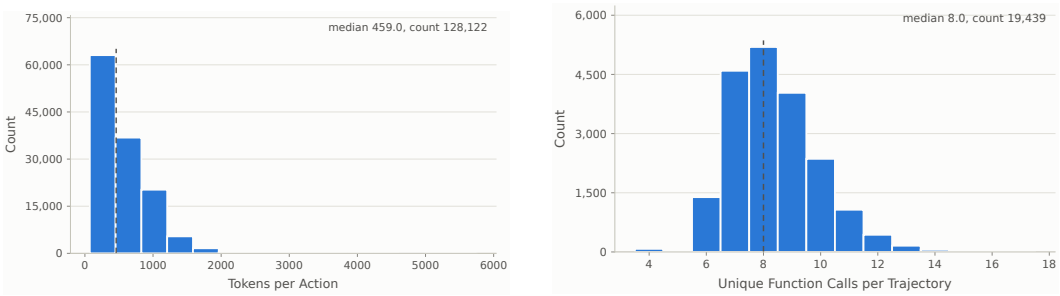
A.4 SynthAPI App and Task Statistics

In this section, we detail the statistics on the overall sets of apps and tasks generated by SynthAPI. Note that these are the statistics from the entire generated population. The experiments in Section 4 are based on subpopulations of this overall data. First, in Figure 4 we visualize statistics of the 64 generated apps and the APIs they expose. Figure 4a shows that apps expose a median of 144 APIs, totaling 10,472 APIs across all apps, of which 6,266 are write and 4,206 are read APIs. Figure 4b shows that apps use a median of 55 distinct data classes, with some apps using over 120. Figure 4c shows that apps implement a median of 81 distinct functionalities. Figure 4d shows that most APIs take between 1 and 10 input arguments. Figure 4e shows that most APIs require at most one ID parameter referencing an existing database entity, with a small number requiring two or three. This indicates that most APIs require composition and first calling other APIs to get the appropriate ID data. Figure 4f shows the distribution of types across all 24,103 API parameters showing there is a diversity of data types beyond strings. Unannotated types refer to dict data types.

B Training Details

For each task set $(\mathcal{A}_i, \mathcal{T}_i)$ we roll out GLM-5.2 (GLM-5 Team, 2026) on every task through the same interface and prompt as the agent to collect data. We use vLLM (Kwon et al., 2023) for sampling and use the default sampling parameters. The GLM expert attempts each task up to 4 times and we save the first trajectory where the task verifier returns True indicating the task was successfully completed. Therefore, each task is only represented with 1 training trajectory in the SFT dataset. When collecting the data we save the reasoning trace from the data collection policy as well for supervision.

Figure 5 visualizes data about the resulting trajectories. The trajectory actions have a median of 459 tokens per action block. The action which is generated after the reasoning includes some preamble



(a) Tokens per action

(b) Unique function calls per trajectory

Figure 5: Statistics of the GLM-5.2 expert trajectories in the SFT dataset, both computed on a subsample of the overall set of training trajectories. **(a)** Distribution of the number of tokens per expert action. **(b)** Distribution of the number of unique function calls per trajectory.

text and then a code block that invokes function calls from the tool library. The actions call a median of 8 tools from the app tool libraries per trajectory, but the count ranges from 4 to 18.

We train the student on the collected data with next-token cross-entropy and apply the loss only to the expert's reasoning text and code blocks. The agent prompt, x, tool results, and all other environment tokens are masked out. We train with bf16, learning rate 2×10^{-5}, and 1% learning rate warmup, max gradient norm of 1.0 and sequence packing. For the 2B model we use pack sequence length 262,144, for 4B we use 262,144 and 2 steps of gradient accumulation, for 9B we use 2 steps of gradient accumulation and pack size 131,072, and for 27B we use pack size 65,536 and gradient accumulation 4 and also train on $2\times$ more nodes than for the other sizes. We train on 8xB200 nodes and use Flash Attention 4 (Zadouri et al., 2026). Unless otherwise specified, we run every experiment on 1 node, and scale the SynthAPI-Big experiments to $2\times$ the number of nodes.

We hold out a random subset of 4k trajectories, spanning different apps, to use as validation data. We evaluate the model on the validation data every 200 updates. The experiments in Section 4.2 and Section 4.3 train for at most 1 epoch over the dataset. However, if the validation loss increases 5 times in a row across successive validation runs, we early stop the training run before the target number of epochs. The SynthAPI-Big experiment in Section 4.4 increases to 4 epochs and uses $2\times$ the amount of compute by scaling the number of nodes but keeps the other settings the same.

C Analysis Details

In all scaling experiments, we fit Equation (2) independently per model size, minimizing the unweighted sum of squared residuals between predicted and observed aggregate score with the trust-region reflective solver in SciPy. We run 512 restarts from initializations drawn uniformly within the parameter bounds and keep the fit with the lowest residual. We hold out the runs in the top 25% of training FLOPs and fit on the remaining runs, so the held-out set lies outside the compute range used for fitting and tests extrapolation. Table 3 shows the fitted parameters.

Table 3: Fitted scaling-law coefficients of Equation (2) by model size.

Model size	$\mathbf{c_0}$	$\mathbf{c_A}$	$\mathbf{b_0}$	$\mathbf{b_T}$	$\boldsymbol{\nu}$
2B	-2.46	0.181	0.54	0.595	20.00
4B	-0.57	0.247	-0.66	0.401	1.67
9B	-0.25	0.236	-0.41	0.495	1.46
27B	0.20	0.392	-9.96	1.475	0.05

D Benchmark and Evaluation Details

In each benchmark environment, we enable the agent to use the same agent interface of Section A.3. Each benchmark's tools are exposed to the agent through the meta-tools of Section A.3, so

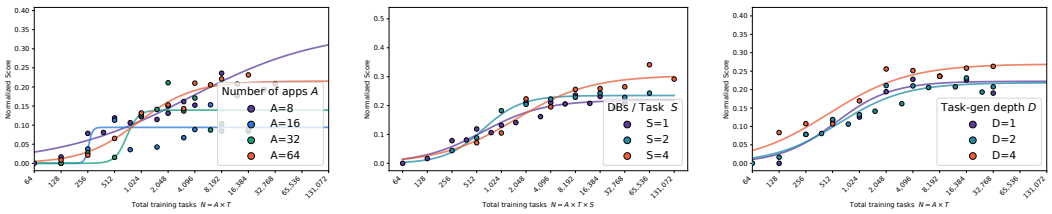
`list_apps`, `list_tools`, and `get_tool_documentation` serve that benchmark's own API documentation. The agent executes the tools by emitting Python code. Any called environment tools are then dispatched to the environment. We use the same agent prompt as used during training.

In AppWorld, we report the task goal completion (TGC) and restrict evaluation to 50 steps maximum. In ASTRA-Bench we restrict the evaluation to only the hardest complexity of each of the three task categories. In BFCL we use v3 and the multi-turn split only with the default benchmark settings. For evaluation, we use the default Qwen3.5 sampling parameters in vLLM. We restrict the maximum number of per-action output tokens to 4096 including the reasoning and the action.

E Additional Results

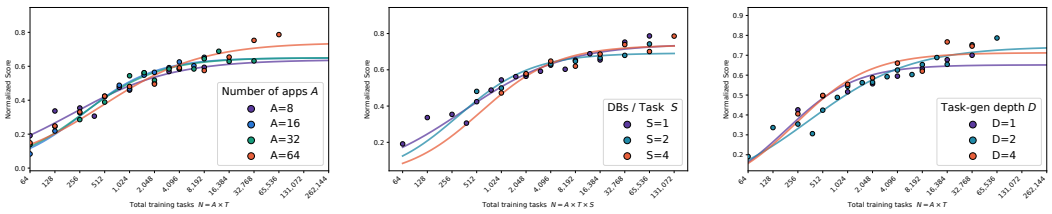
E.1 Additional Analysis

We show all the per model size app-to-task allocation plots below. We also report the fit quality of the scaling law in Equation (2) for each student size, both on the runs used for fitting (in-sample) and on all held-out runs in the top 25% of training FLOPs (extrapolation), in units of the normalized aggregate score. For the 2B student, the in-sample RMSE is 0.043 and MAE is 0.033, and the extrapolation RMSE and MAE are both 0.004. For the 4B student, the in-sample RMSE is 0.050 and MAE is 0.039, and the extrapolation RMSE is 0.066 and MAE is 0.051. For the 9B student, the in-sample RMSE is 0.050 and MAE is 0.038, and the extrapolation RMSE is 0.033 and MAE is 0.026. For the 27B student, the in-sample RMSE is 0.076 and MAE is 0.049, and the extrapolation RMSE is 0.037 and MAE is 0.028.



(a) App vs. task allocation (b) Database breadth (c) Database depth

Figure 6: Allocation experiment for the 2B model.



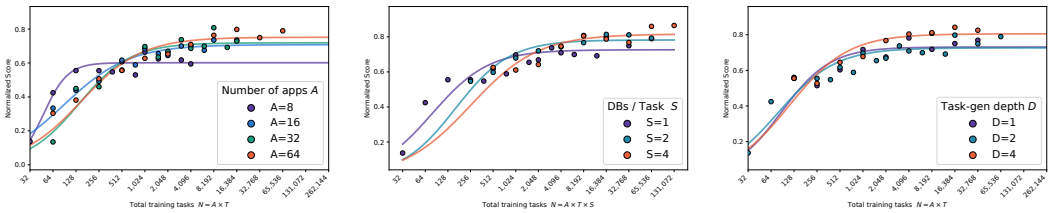
(a) App vs. task allocation (b) Database breadth (c) Database depth

Figure 7: Allocation experiment for the 4B model.

E.2 Benchmark Results Breakdown

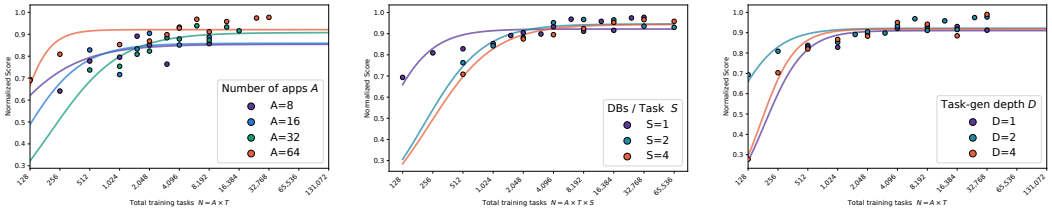
Table 1 reports a single aggregate number per benchmark and model size. Tables 4 to 6 give the full breakdown behind those aggregates: both AppWorld splits, and the per-category scores on ASTRA-Bench and the SynthAPI holdout.

E.3 Results on Another Data Synthesis Pipeline



(a) App vs. task allocation (b) Database breadth (c) Database depth

Figure 8: Allocation experiment for the 9B model.



(a) App vs. task allocation (b) Database breadth (c) Database depth

Figure 9: Allocation experiment for the 27B model.

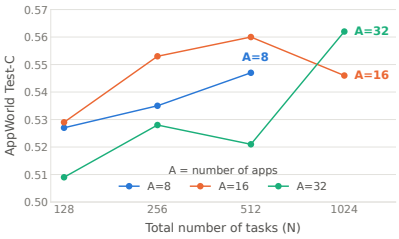


Figure 10: Comparing the allocation between apps and tasks per-app on another data synthesis pipeline SimSAT (Lee et al., 2026).

Here, we show that the findings we observe in Section 4.2—on how tool diversity (measured by number of apps) affects the scaling behavior as we increase the total number of tasks—hold not only on our SynthAPI pipeline, but are also valid on another data synthesis pipeline. Specifically, we validate our findings with another synthetic tool-use dataset, SimSAT (Lee et al., 2026). SimSAT contains synthetic tasks and their corresponding tool-use trajectories generated from 50 different apps. Under fixed task budgets of $N = \{128, 256, 512, 1024\}$, we sample various sets of tasks generated from varying number of apps ($A = \{8, 16, 32\}$), and study how downstream performance changes. As shown in Figure 10, we see that under lower task budgets with $N = \{128, 256, 512\}$, it is favorable to generate more tasks per app ($A = 32$ generally underperforms compared to $A = \{8, 16\}$). However, with a higher task budget at $N = 1024$, we see an opposite trend that generating tasks from more apps unlocks better performance, comparing $A = 32$ to $A = 16$. This aligns with our observation on SynthAPI, where high tool diversity matters as we scale the number of training tasks.

E.4 Scaling Law Fit Quality

	Qwen3.5-4B		Qwen3.5-9B		Qwen3.5-27B	
Approach	Test-N	Test-C	Test-N	Test-C	Test-N	Test-C
Zero-shot	47.5	14.7	45.4	26.4	81.8	61.5
SimSAT	66.8	58.8	74.5	65.4	-	-
Nemotron	26.0	17.5	31.9	20.8	-	-
ToolACE	20.2	10.9	18.6	12.4	-	-
ToolAlpaca	25.7	17.3	24.4	15.7	-	-
SynthAPI-SubOpt	39.3	38.4	56.5	53.0	78.6	73.8
SynthAPI-Opt	47.0	45.0	58.2	56.2	81.0	74.1
SynthAPI-Big	**81.0**	**66.4**	**83.3**	**77.0**	**90.6**	**81.5**

Table 4: Full AppWorld results on the Test-Normal (Test-N) and Test-Challenge (Test-C) splits. The SimSAT, ToolACE, and ToolAlpaca results are from Lee et al. (2026).

Model	Approach	Ref.	Func.	Info.	Overall
Qwen3.5-4B	Zero-shot	55.5	33.5	60.3	57.4
	Nemotron	50.1	27.5	58.6	54.4
	SynthAPI-SubOpt	69.7	48.1	74.4	71.1
	SynthAPI-Opt	71.4	51.0	74.8	72.3
	SynthAPI-Big	**73.7**	**53.2**	**76.3**	**74.2**
Qwen3.5-9B	Zero-shot	69.5	50.3	74.2	71.8
	Nemotron	64.6	47.2	70.4	67.7
	SynthAPI-SubOpt	70.4	49.1	78.2	74.2
	SynthAPI-Opt	71.9	52.1	78.9	75.5
	SynthAPI-Big	**75.5**	**53.2**	**81.1**	**76.8**
Qwen3.5-27B	Zero-shot	79.1	62.4	82.7	80.2
	Nemotron	-	-	-	-
	SynthAPI-SubOpt	74.1	55.7	80.4	77.3
	SynthAPI-Opt	78.6	58.8	81.3	79.3
	SynthAPI-Big	**82.8**	**64.7**	**85.4**	**82.3**

Table 5: Full ASTRA-Bench results broken down by task category. Note that in ASTRA-Bench, a single task can count for multiple categories, thus the overall score is not the average of the individual columns.

Model	Approach	Shop.	Msg.	Fin.	Music	Avg.
Qwen3.5-4B	Zero-shot	0.0	0.0	1.0	0.0	0.3
	Nemotron	0.0	0.0	1.0	3.0	1.0
	SynthAPI-SubOpt	50.0	51.0	**60.0**	44.0	51.3
	SynthAPI-Opt	**64.0**	**74.0**	36.0	**50.0**	**56.0**
	SynthAPI-Big	60.0	72.0	32.0	48.0	53.0
Qwen3.5-9B	Zero-shot	1.0	1.0	3.0	1.0	1.5
	Nemotron	0.0	1.0	0.0	1.0	0.5
	SynthAPI-SubOpt	64.0	72.0	40.0	48.0	56.0
	SynthAPI-Opt	71.0	**79.0**	47.0	**55.0**	63.0
	SynthAPI-Big	**76.0**	76.0	**52.0**	52.0	**64.0**
Qwen3.5-27B	Zero-shot	8.0	35.0	15.0	22.0	20.0
	Nemotron	-	-	-	-	-
	SynthAPI-SubOpt	72.0	76.0	56.0	52.0	64.0
	SynthAPI-Opt	74.0	**78.0**	64.0	**56.0**	68.0
	SynthAPI-Big	**80.0**	77.0	**68.0**	54.0	**69.8**

Table 6: Full SynthAPI-Holdout results broken down by app category.

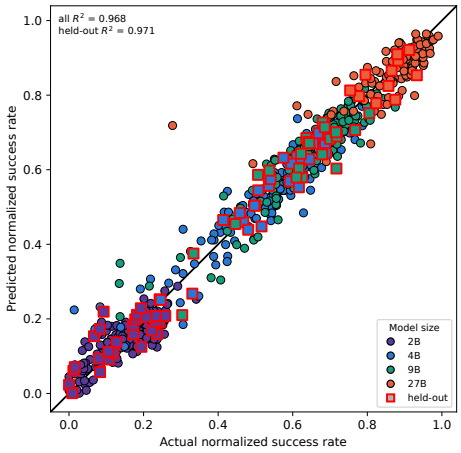


Figure 11: Fit quality of the joint scaling law. Each point compares the performance predicted by Equation (2) with the observed aggregate normalized benchmark score for one experimental run, colored by student model size, with runs held out from the fit (the top 25% of training FLOPs) outlined in red. The diagonal line indicates perfect prediction.