ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Final Project

Dissertation

Linalg-Zero: Distilling Neurosymbolic Reasoning for Linear Algebra in Small Language Models

Author:
Razvan F. Vasile
Degree Program:
Laurea Magistrale in
Ingegneria e Scienze
Informatiche

Supervisor:
Prof. Matteo Ferrara

February 24, 2026

Contents

1 Introduction 1
1.1 Overview . . . 1
1.2 Related Work . . . 4
1.3 Contributions and Outline . . . 5

2 Definitions and Background 7
2.1 Large Language Models and Tokenization . . . 7
2.1.1 LMs as Conditional Sequence Models . . . 7
2.1.2 Tokenization and Structured Outputs . . . 8
2.1.3 Decoding and Trajectory Reliability . . . 10
2.2 Data Generation and Distillation . . . 11
2.2.1 Synthetic Dataset Generation . . . 12
2.2.2 Knowledge Distillation and Rejection Sampling . . . 13
2.3 Fine-Tuning and Alignment . . . 14
2.3.1 Supervised Fine-Tuning (SFT) . . . 15
2.3.2 Reinforcement Learning (GSPO) . . . 15
2.4 Evaluation Methodology . . . 16
2.5 Summary . . . 17

3 Dataset Generation 18
3.1 Related Work . . . 18
3.2 Overview . . . 19
3.2.1 Taxonomy & Constraints . . . 19
3.3 Optimizations/Analysis . . . 21
3.3.1 Design patterns . . . 21
3.3.2 Entropy Based Generation . . . 22
3.4 Experiments . . . 25
3.4.1 Curriculum Validation . . . 25
3.4.2 Generate Splits . . . 27
3.5 Conclusion . . . 28

4 Distillation 29
4.1 Related Work . . . 29
4.2 Overview . . . 30
4.3 Model Architecture . . . 31
4.4 Optimizations/Analysis . . . 33

4.5 Experiments . . . 34
  4.5.1 Dataset Cleaning and Filtering . . . 35
  4.5.2 Infrastructure and Scalability . . . 36
4.6 Conclusion . . . 37

5 SFT **38**
5.1 Related Work . . . 38
5.2 Overview . . . 39
5.3 Model Architecture . . . 40
5.4 Analysis . . . 41
5.5 Experiments . . . 42
  5.5.1 Experimental Setup . . . 42
  5.5.2 Ablation: No Special Tokens for Interface Tags . . . 43
  5.5.3 SFT Loss and Accuracy Trends . . . 44
5.6 Conclusion . . . 45

6 Group Sequence Policy Optimization **46**
6.1 Related Work . . . 46
6.2 Overview . . . 47
6.3 Architecture . . . 48
6.4 Optimizations/Analysis . . . 50
  6.4.1 Training Stability . . . 50
6.5 Experiments . . . 51
  6.5.1 Curriculum . . . 51
  6.5.2 Results . . . 52
6.6 Conclusion . . . 55

7 Conclusion **56**
7.1 Ethical Considerations . . . 56
7.2 Future Work & Limitations . . . 57
7.3 Conclusions . . . 57

Appendices **66**
Appendices . . . 66
A.1 Released Artifacts . . . 66
  A Deliverables . . . 66
A.2 Generation . . . 67
  A Linear Algebra Primitives . . . 67
  B Problem-Type Taxonomy . . . 68
  C Representative Generated Examples . . . 69
  D Generation Algorithm . . . 70
  E Generation Schematics . . . 71
A.3 Distillation . . . 72
  A System Prompt . . . 72
  B Tool-Use Trace Examples . . . 73
A.4 Reinforcement Learning . . . 74
  A Reward Function . . . 74

B GRPO/SFT Side-by-Side Comparison 75
C Malformed Evaluation Examples 79

List of Figures

1.1 A minimal multi-turn tool-use example 2
1.2 End-to-end process pipeline 3

2.1 GSPO training loop 16

3.1 Dataset generation workflow 19
3.2 Example synthetic dataset record 20
3.3 Dataset generation design patterns 22
3.4 Entropy-controlled sampling strategy 24
3.5 Monte Carlo entropy calibration loop 27
3.6 Dataset composition by difficulty tier 28

4.1 Distillation workflow schematic 30
4.2 Four-phase tool-use turn 31
4.3 Rejection-sampling distillation loop 33
4.4 Trace length distributions (raw vs. filtered) 36
4.5 RunPod compute infrastructure for distillation 37

5.1 SFT workflow schematic 39
5.2 NST ablation loss curves 43
5.3 SFT training dynamics 44

6.1 GRPO workflow diagram 47
6.2 Rollout generation pipeline 48
6.3 GSPO training loop 49
6.4 Curriculum scheduling during GRPO 51
6.5 GSPO training dynamics under a frozen head 52
6.6 GRPO training diagnostics 53

D.1 Generation pipeline for synthetic problems 70
E.1 Problem construction for multi-step questions 71

List of Tables

2.1 Standard CoT vs. tool-integrated reasoning 12

3.1 Linear Algebra tool definitions 21
3.2 Entropy Calibration Results 26

4.1 Two-step tool-use trace 32
4.2 Distillation reliability and filtering breakdown 35

5.1 SFT Hyperparameters 42
5.2 Ablation: no special tokens 43
5.3 Merged checkpoint metrics 45

6.1 Eval error breakdown (SFT vs GSPO) 54
6.2 Test results (evaluation set) 55

A.1 Project deliverables 66
B.1 Problem-type taxonomy 68
A.1 System prompt 72
B.1 Tool-use trace examples (appendix) 73
B.1 Matched SFT vs GRPO tool-use traces (1) 76
B.2 Matched SFT vs GRPO tool-use traces (2) 78
C.1 Representative eval traces by failure type 85

Abstract

The main drawback of using generative AI models for advanced mathematics is that they are not deterministic logical reasoning engines. Because neural models are probabilistic, they can produce outputs that look convincing while still containing subtle errors that are hard to detect without careful algorithmic verification.

Symbolic methods can largely address this issue by performing exact, deterministic calculations through code execution. However, solving a full problem typically requires an explicit plan: the correct sequence of tool invocations and the dependencies between intermediate results must be specified, and this often requires human intervention. Our premise is that this planning burden can be reduced by using neural models to orchestrate tool use, while symbolic solvers provide the exact computation.

In this thesis, we demonstrate an end-to-end workflow that combines neural models with symbolic solvers to solve linear algebra problems through tool-use interactions, in a controlled, verifiable setting with a small, audited tool library. Our results show that, starting from a small pre-trained base model (Qwen2.5-3B), it is possible to achieve 90% test-set accuracy (verifier-checked on a fixed held-out evaluation set) on problem traces requiring up to three tool interactions.

The pipeline includes synthetic dataset generation, distillation, supervised fine-tuning (SFT), and reinforcement learning via Group Sequence Policy Optimization (GSPO). Using parameter-efficient fine-tuning (LoRA) and on-demand cloud GPUs, the full pipeline is reproducible within a $75 budget. This provides a concrete recipe for practitioners to train self-hostable tool-using models, and a pedagogical blueprint for students learning to build tool-calling agents beyond prompt engineering. Code and configs are available at `https://github.com/atomwalk12/linalg-zero`.

Acknowledgements

I thank my supervisor, Prof. Matteo Ferrara, for his patience, guidance, and steady encouragement throughout this research.

I also thank my family for their support during this project and my degree; their encouragement has been a constant source of inspiration.

Chapter 1

Introduction

1.1 Overview

> *"The hardest problems we have to face do not come from philosophical questions about whether brains are machines or not. There is not the slightest reason to doubt that brains are anything other than machines with enormous numbers of parts that work in perfect accord with physical laws. As far as anyone can tell, our minds are merely complex processes. The serious problems come from our having had so little experience with machines of such complexity that we are not yet prepared to think effectively about them. Our present-day technology does not yet allow us to study the brain cells of higher animals while they're actually working and learning."*
>
> — Marvin Minsky, *The Society of Mind (1986)*

Epigraph. Minsky's observation motivates the design philosophy of this thesis: the core difficulty is not that complex learning machines are mysterious in principle, but that they are hard to study systematically. Therefore, we focus on a controlled, verifiable setting where the model must interact with tools under a strict interface contract, so failure modes are observable as parse, execution, or correctness errors rather than hidden in the semantics of the model's natural language output. Consequently, the pipeline in this report is built to turn "reasoning" into an auditable interaction trace.

Moreover, we treat this pipeline as a testbed for studying how a complex learning system can evolve into reliable, tool-mediated problem-solving, because linear algebra is narrow enough to be verifiable while still rich enough to require multi-step decisions. Therefore, what was difficult in Minsky's era – assessing learning dynamics inside a complex machine systematically – becomes feasible here through controlled interfaces and replayable traces.

Thesis Aim. This thesis studies how to train a small pretrained language model – `Qwen2.5-3B` [55] – to solve a controlled set of linear-algebra problems by producing *executable tool-use trajectories* rather than unverified free-form text. Therefore, unlike traditional answer-only evaluation that checks only the final generated result, we treat format adherence and tool-call validity as first-class constraints: the trace must remain executable turn-by-turn, and its final answer must match a deterministic verifier.

```
User: A = [[1, 2], [3, 4]]. Compute det(A^T).
Assistant: <tool_call>{
  "name": "transpose",
  "arguments": {"matrix": [[1, 2], [3, 4]]}
}</tool_call>
Tool: [[1, 3], [2, 4]]
Assistant: <tool_call>{
  "name": "determinant",
  "arguments": {"matrix": [[1, 3], [2, 4]]}
}</tool_call>
Tool: -2
Assistant: <answer>-2</answer>
```

Figure 1.1: A minimal multi-turn tool-use trajectory for computing det($A^\top$) by calling `transpose` and then `determinant`.

Motivation (Verifiable Domain). Linear algebra is a useful domain because its intermediate computations are concrete objects – like a matrix of size $N \times M$ represented as a Python list-of-lists – and its final answers can be checked automatically. Moreover, when the model is allowed to call a small tool library, numeric correctness becomes deterministic execution, which lets us measure progress without human graders.

Concrete Anchor. To make the setting tangible, consider the task "compute det($A^\top$)" for $A = \begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}$. In Figure 1.1, initially, the model must emit a syntactically valid tool call. Moreover, it must incorporate the tool output into its next action. Only then can it terminate with a verifier-checkable answer. Consequently, evaluation is not "does the model sound right?", but "does the trajectory remain executable until the final answer matches the ground truth?".

The Bottleneck. Tool use shfits errors from "wrong number" to "invalid interface". Paradoxically, although the underlying computations are trivial once delegated to a deterministic library, the model must still plan and execute a multi-step interaction. Because of this, long-horizon tool use becomes probabilistic: if a single-turn format adherence rate is p, then a trajectory requiring N consecutive valid turns has an upper-bound success rate near p^N. Therefore, we treat tool use as four coupled phases – planning, selection, invocation, and response integration – and we track interface adherence as a core constraint alongside correctness.

Pipeline Overview. We address the tool-use bottleneck with an end-to-end pipeline summarized in Figure 1.2. Moreover, each panel in the diagram maps directly onto the method chapters (Generation: Chapter 3; Distillation: Chapter 4; Supervised Fine-Tuning (SFT): Chapter 5; Group Relative Policy Optimization (GRPO): Chapter 6). Therefore, the figure is best read as a sequence of produced artifacts – datasets, traces, and checkpoints – where each stage produces the object consumed by the next.

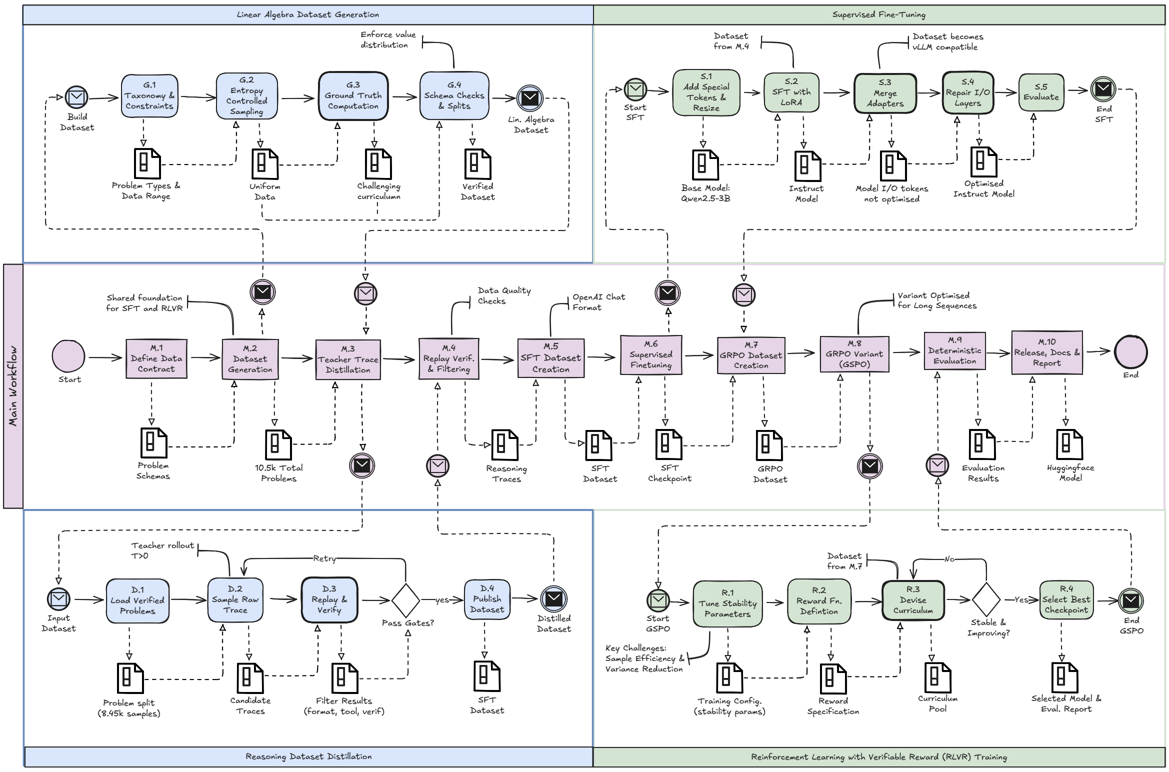

Linear Algebra Dataset Generation
Supervised Fine-Tuning
Enforce value distribution
Build Dataset
G.1 Taxonomy & Constraints
G.2 Entropy Controlled Sampling
G.3 Ground Truth Computation
G.4 Schema Checks & Splits
Lin. Algebra Dataset
Problem Types & Data Range
Uniform Data
Challenging curriculumn
Verified Dataset
Dataset from M.4
Dataset becomes vLLM compatible
Start SFT
S.1 Add Special Tokens & Resize
S.2 SFT with LoRA
S.3 Merge Adapters
S.4 Repair I/O Layers
S.5 Evaluate
End SFT
Base Model: Qwen2.5-3B
Instruct Model
Model I/O tokens not optimised
Optimised Instruct Model
Main Workflow
Shared foundation for SFT and RLVR
Data Quality Checks
OpenAI Chat Format
Variant Optimised for Long Sequences
Start
M.1 Define Data Contract
M.2 Dataset Generation
M.3 Teacher Trace Distillation
M.4 Replay Verif. & Filtering
M.5 SFT Dataset Creation
M.6 Supervised Finetuning
M.7 GRPO Dataset Creation
M.8 GRPO Variant (GSPO)
M.9 Deterministic Evaluation
M.10 Release, Docs & Report
End
Problem Schemas
10.5k Total Problems
Reasoning Traces
SFT Dataset
SFT Checkpoint
GRPO Dataset
Evaluation Results
Huggingface Model
Teacher rollout T>0
Retry
Input Dataset
D.1 Load Verified Problems
D.2 Sample Raw Trace
D.3 Replay & Verify
Pass Gates?
yes
D.4 Publish Dataset
Distilled Dataset
Problem split (8.45k samples)
Candidate Traces
Filter Results (format, tool, verif)
SFT Dataset
Dataset from M.7
No
Start GSPO
R.1 Tune Stability Parameters
R.2 Reward Fn. Defintion
R.3 Devise Curriculum
Stable & Improving?
Yes
R.4 Select Best Checkpoint
End GSPO
Key Challenges: Sample Efficiency & Variance Reduction
Training Config. (stability params)
Reward Specification
Curriculum Pool
Selected Model & Eval. Report
Reasoning Dataset Distillation
Reinforcement Learning with Verifiable Reward (RLVR) Training

Figure 1.2: End-to-end process pipeline. Artifacts flow from verified synthetic problems to executable traces, to an interface-compliant SFT checkpoint, and finally to GRPO training.

- **Main Workflow (M.1 - M.10).** The central diagram is the end-to-end workflow that connects Generation, Distillation, SFT, and GRPO. Moreover, all phases measure progress under a single parse → execute → verify loop.
- **Generation (G.1 - G.5).** Firstly, we define the task categories and constraints. Then, entropy-based sampling controls numeric diversity, cross-component composition builds multi-step tool chains, curriculum validation optimises entropy ranges to ensure a stable distribution, and split generation produces the final verified dataset.
- **Distillation (D.1 - D.4).** Secondly, verified problems are converted into multi-turn trajectories and executed. Because of this, malformed tags or non-replayable tool calls are rejected immediately, yielding a clean reasoning dataset.
- **SFT (S.1 - S.5).** Thirdly, we make the tool interface learnable by registering the control delimiters (i.e. `<think>`, `<tool_call>` and `<answer>`) as special tokens, training adapters, and compiling a standalone checkpoint via Merge-Repair-Freeze. The result is a vLLM-compatible model, used during the GRPO phase.
- **GRPO (R.1 - R.4).** Finally, we optimize correctness under the same parse-execute-verify protocol. Then, rewards are defined over success metrics (correctness, format and executability), stability parameters are tuned, a curriculum is applied to stabilize learning, and checkpoints are selected under deterministic evaluation.

Research Questions. This thesis is organized around four questions anchored in the pipeline artifacts:

- How can we generate a verified synthetic linear-algebra curriculum with controllable difficulty and coverage? (Chapter 3)
- How can we distill verified problems into multi-turn, executable tool-use trajectories under a strict format contract? (Chapter 4)
- What does SFT contribute beyond correctness – in particular, how does it stabilize interface reliability under a strict tool-calling contract? (Chapter 5)
- Can GRPO improve correctness beyond SFT while preserving format stability, and what learning dynamics emerge under this trade-off? (Chapter 6)

Evaluation. We evaluate models in the same way they are used: parse a turn, execute a tool call if present, append the tool output, and continue until a final answer is produced. Therefore, we report both outcomes (verifier-checked correctness) and constraints (format validity, tool-call validity), because a correct plan is useless if it cannot be expressed as an executable trace. Moreover, accuracy is reported on a fixed held-out test split under greedy decoding, so it reflects interface stability rather than sampling luck. On the other hand, sampling is reserved for training-time rollouts during reinforcement learning.

Scope and Limits. This work does not target open-domain mathematics, proof generation, or tool frameworks. On the other hand, it makes a deliberate trade: by restricting the setting to synthetic, verifiable linear algebra with bounded matrix sizes and a small tool vocabulary, we can study tool-use learning dynamics systematically and report results using deterministic, executable checks without relying on subjective grading.

1.2 Related Work

Tool-Augmented Reasoning. Tool use makes reasoning an action-selection problem under a fixed interface format: the model must decide *what* to compute and also *how* to encode the computation as structured output. Therefore, this thesis is inspired by recent tool-use surveys that argue the literature remains fragmented and lacks a unified tool-calling framework [76]; accordingly, we align with program-aided [19] and tool-interaction [81] approaches, but emphasize multi-turn execution as an evaluation constraint rather than assessing final-answer generation alone.

Synthetic Curriculum. Synthetic datasets are especially useful in verifiable domains because generators can produce ground truth at scale and can enforce difficulty via composition. Firstly, we investigate synthetic data generation by following the survey analysis in [43]. Moreover, we find it necessary to build an explicit difficulty curriculum to enable small models to learn effectively. Because of this, we control difficulty in two ways: entropy-controlled sampling constrains the generator's distributions, and we construct problems that require 1, 2, or 3 tool calls to solve. Consequently, these constraints keep numeric ranges fixed within predefined bounds, as in [58]. On the other hand, our work extends existing work on symbolic tasks from other domains (e.g. arithmetic, calculus, probability) to a tool-mediated linear algebra curriculum.

Distillation. Distillation provides a direct bridge from verified problems to chain-of-thought solutions. Because of this, we distill not only answers but structured multi-turn traces, and we treat verification as a filter rather than a post-hoc cleanup step, which ensures that the resulting trajectories are valid inputs for downstream supervised fine-tuning (SFT) [50]. Moreover, our implementation uses the `distilabel` [90] library to orchestrate and validate the distillation pipeline at scale.

RLVR and Prior Work. Reinforcement Learning from Verifiable Rewards [75] (RLVR) is well-suited to domains where correctness can be checked automatically. Moreover, recent open-source reasoning releases have made this training approach practically accessible to a wider research community [12]. Furthermore, our end-to-end pipeline follows reproducible open implementations that emphasize practical training recipes, such as TinyZero [51], Nano Aha! Moment [34], and Open R1 [30].

GRPO and the SFT Prerequisite. Finally, replication surveys of the DeepSeek-R1 pipeline emphasize that the original recipe starts from a *base* model rather than an instruction-tuned model [84]; therefore, we adopt the same starting point and use GRPO-style group-relative rewards with PPO-style stabilization to preserve the training dynamics [60, 64]. Nonetheless, because base models do not reliably follow instruction formats, we introduce an a priori SFT phase to make the tool-calling interface learnable; in practice, producing multi-turn, schema-valid tool calls is a more challenging task than single-turn responses. Consequently, the ablation in Chapter 5 shows that SFT is a prerequisite for effective tool learning in our approach.

1.3 Contributions and Outline

Released Artifacts. The pipeline produces concrete, reproducible artifacts in the form of datasets and checkpoints. Therefore, Appendix A lists the Hugging Face Hub releases (Table A.1).

Contributions. The main contributions of this thesis are:

- An end-to-end, reproducible tool-use pipeline, where every stage is evaluated under the same interface contract (parse → execute → verify), making multi-turn executability a primary evaluation constraint across all stages (Chapters 1 – 6).
- An entropy-controlled synthetic linear-algebra generator that produces verifiable problems with controlled compositional depth (Chapter 3).
- A distillation pipeline that converts problems into multi-turn tool-use conversations under a strict tag format (Chapter 4).

- An SFT stage that instruction-tunes the `Qwen2.5-3B` base model [55] for tool use by introducing special format tokens, together with a format-preserving checkpoint that produces a GRPO-compatible standalone checkpoint (Chapter 5).
- A GRPO training setup that improves verifier-checked correctness beyond SFT while analyzing stability limits induced by extended, multi-turn tool-use rollouts (Chapter 6).

Outline. The remainder of this thesis is organized as follows. Chapter 2 introduces the background needed for tool-using language models and verifier-based evaluation. Chapters 3 and 4 describe dataset generation and distillation. Chapter 5 presents the SFT stage and associated format constraints. Chapter 6 presents GRPO experiments and stability considerations. Finally, Chapter 7 summarizes limitations, ethical considerations, and future work.

Chapter 2

Definitions and Background

Overview. This chapter reviews the background needed for this thesis: LLMs and tokenization, tool-calling, and post-training methods. For a practical introduction to Transformers (including tokenization and fine-tuning), see *Natural Language Processing with Transformers* [69]. For terminology and post-training objectives, see Lambert's *Reinforcement Learning from Human Feedback* [38] (RLHF).

2.1 Large Language Models and Tokenization

2.1.1 LMs as Conditional Sequence Models

Overview. This subsection defines a decoder-only Transformer as an autoregressive policy under causal masking. Because this thesis studies *multi-turn tool-use trajectories* (parse → execute → verify), this section clarifies why tool calls are generated token-by-token and why format validity compounds across turns.

Definition 2.1.1 (Autoregressive Factorization). An autoregressive language model represents a joint distribution over a token sequence by factorizing it into next-token conditionals:

$$p_\theta(x_{1:T}) = \prod_{t=1}^{T} p_\theta(x_t \mid x_{<t}). \tag{2.1}$$

Definition 2.1.2 (Transformer (Decoder-Only)). A Transformer maps a token sequence into contextualized hidden states by alternating self-attention and position-wise feed-forward layers [70]. In a decoder-only model, self-attention is causal, implying that hidden state at time t depends only on the prefix $x_{\leq t}$.

Definition 2.1.3 (Causal Masking). Causal masking enforces autoregressive generation by preventing attention from reading future tokens [70]. Concretely, when computing attention at position t, the model is restricted to attend only to positions $\leq t$.

Definition 2.1.4 (Teacher Forcing). Teacher forcing trains a language model by always feeding it the correct previous tokens during training. Concretely, when predicting token x_t, the model conditions on the true prefix $x_{<t}$; however, at inference time it produces text based on its own generated prefix instead [5].

Link to distillation. Therefore, even when our training sequences come from distillation (Section 2.2.2), we still apply teacher forcing within each sequence during SFT.

Definition 2.1.5 (Next-token negative Log-Likelihood (NLL))**.** Given a tokenized sequence $x_{1:T}$, decoder-only pre-training minimizes the negative log-likelihood of the next token:

$$\mathcal{L}_{\mathrm{NLL}}(\theta) = -\sum_{t=1}^{T} \log p_\theta(x_t \mid x_{<t}). \tag{2.2}$$

Definition 2.1.6 (Tool-Use Trajectory)**.** A tool-using solution can be modeled as a trajectory of turns:

$$\text{prompt} \rightarrow \text{tool call} \rightarrow \text{tool output} \rightarrow \cdots \rightarrow \text{final answer.}$$

Unlike single-shot question answering, intermediate actions must be valid, because later steps depend on the tool outputs they produce [81, 59].

Definition 2.1.7 (ReAct (Reasoning + Acting))**.** ReAct is a tool-use prompting pattern where the model alternates between brief reasoning and explicit actions, treating each tool output as an observation for the next step [81]. Concretely, for a linear-algebra task, the model can call SymPy to compute a determinant, then use that returned scalar to select the next operation.

Definition 2.1.8 (Executable Trace)**.** An executable trace is a serialized multi-turn output whose tool-call blocks can be parsed and replayed end-to-end (e.g. tags plus JSON arguments). Therefore, this makes evaluation reliable and separates errors into parsing failures, tool-execution errors, or verifier mismatches.

2.1.2 Tokenization and Structured Outputs

Overview. This subsection explains how tokenization maps text to token IDs and how generation is constrained by the vocabulary via `embed_tokens` and the `lm_head`. Therefore, it shows that emitting exact control tags (e.g. `<tool_call>`) is necessary for structured outputs to remain parseable.

Definition 2.1.9 (Tokenizer and Tokenization)**.** Tokenization converts raw text into token IDs from a fixed vocabulary $\mathcal{V}$ [62] (typically learned via Byte-Pair Encoding (BPE) [73] or SentencePiece [35]), so the model operates on discrete symbols; for example, a delimiter like `<tool_call>` may be a single token or split across multiple tokens.

Therefore, tokenization is a format constraint: if a delimiter like `<tool_call>` is split across multiple tokens, the model must emit the full sequence correctly every time, and multi-turn errors compound. It is possible to overcome this limitation by registering special tokens.

Definition 2.1.10 (Input Embeddings – `embed_tokens`)**.** The token embedding layer maps each vocabulary item to a learnable vector. Concretely, `embed_tokens` is an embedding matrix $E \in \mathbb{R}^{V \times H}$ that converts a token ID $i \in \{1, \ldots, V\}$ into a hidden vector $E_i \in \mathbb{R}^H$.

Definition 2.1.11 (LM Head (Output Projection))**.** The model's final hidden state has dimension H, but the output space is the vocabulary of size $V = |\mathcal{V}|$. The `lm_head` is a learned linear map in $\mathbb{R}^{H \times V}$ that converts hidden states into logits over $\mathcal{V}$, constraining generation to the current vocabulary.

Definition 2.1.12 (Vocabulary Expansion & Special Tokens)**.** Vocabulary expansion adds new symbols to the vocabulary $\mathcal{V}$, including control delimiters such as <`think`>, <`too_call`>, and <`answer`>. Registering these delimiters as special tokens prevents subword splitting and improves format reliability for tool calling.

However, adding tokens requires resizing `embed_tokens` and the `lm_head`, which can temporarily perturb the model's output distribution until the new parameters are realigned during fine-tuning.

Definition 2.1.13 (Control Tags)**.** Control delimiters are special tokens (e.g. <`think`>, <`answer`>, <`tool_call`>) that mark the boundaries of structured blocks within the model output, so an external parser can extract text reliably.

Definition 2.1.14 (Vocabulary Resize and Realignment)**.** Vocabulary realignment refers to the engineering steps required after adding special tokens. Because vocabulary expansion modifies both `embed_tokens` and the `lm_head`, the new parameters can modify the model's output distribution until they are realigned.

Example 2.1.15 (Vocabulary Resize and Realignment)**.** Consider a base model with a tokenizer vocabulary of size V and hidden size H, so `embed_tokens` has shape $V \times H$ and the `lm_head` has shape $H \times V$. Therefore, generation can only assign probabilities to the existing V token IDs, that is, choosing one entry from a length-V list of logits.

Now suppose we add K new special tokens (e.g. <`tool_call`>, <`answer`>) and the tokenizer does *not* already contain K unused slots reserved for expansion. Because of this, we must resize the embedding and head to match the new vocabulary size $V' = V + K$, which introduces new rows/columns that are typically randomly initialized. Consequently, even if all other model weights are unchanged, the altered `embed_tokens`/`lm_head` can perturb the output distribution and degrade formatting.

The purpose of SFT is to tune the newly introduced, randomly initialized parameters associated with the special tokens, so they align with their intended semantics.

Definition 2.1.16 (Chat Template)**.** A chat template is a procedure that converts a multi-turn conversation (roles + messages) into a single token sequence for the model [69]. It defines the role markers and separators, so the model can distinguish user queries, its own responses, and tool-call or tool-output instructions during training and inference.

Example 2.1.17 (Chat template serialization)**.** Suppose a conversation contains the following turns:

- `user`: `Compute` A^{-1} `for A=[[2,1], [1,1]].`
- `assistant`: <`tool_call`>`{...}`</`tool_call`>
- `tool`: `[[1,-1], [-1,2]]`
- `assistant`: `Here is` A^{-1}`: [[1,-1], [-1,2]].`

A chat template merges these turns into a single string by inserting role markers and separators, e.g.

```
<|user|> Compute A^{-1} for A=[[2,1], [1,1]].
<|assistant|> <tool_call>{...}</tool_call>
<|tool|> [[1,-1], [-1,2]]
<|assistant|> Here is A^{-1}:  [[1,-1], [-1,2]].
```

The tokenizer then maps this string into token IDs (and `embed_tokens` maps IDs into vectors), so the model receives one unified sequence with explicit role boundaries rather than separate messages.

Definition 2.1.18 (Format Contract)**.** The format contract [69] is the requirement that the model's output matches the system's expected schema (delimiters, tags, and argument structure). For tool use, this typically means emitting valid JSON arguments.

Definition 2.1.19 (Stop Conditions)**.** Stop conditions [69] are termination rules applied during decoding, such as emitting an end-of-sequence token, reaching a maximum response length, or matching an explicit stop sequence. In tool-using settings, carefully chosen stop strings (i.e. `</tool_call>`, `</answer>`) prevent the model from generating extra tokens beyond the end-of-turn tokens, keeping outputs parseable.

2.1.3 Decoding and Trajectory Reliability

Overview. This subsection defines decoding choices (the trade-off between exploration and validity), and explains how failure probability compounds with trajectory length.

Note (Setup)**.** At inference time, the `lm_head` maps the current hidden state to logits over the vocabulary. These logits define a probability distribution, and decoding specifies how the next token is chosen – either deterministically (greedy) or stochastically (sampling).

Definition 2.1.20 (Greedy Decoding)**.** Greedy decoding is a deterministic inference strategy that selects the most probable token at each step, i.e., $x_t = \arg\max_v p_\theta(v \mid x_{<t})$ [69]. It is often effective for maintaining rigid syntax (e.g. valid JSON), but it can reduce exploration by repeatedly choosing locally likely tokens.

Definition 2.1.21 (Temperature Sampling)**.** Temperature sampling introduces stochasticity by sampling from a softened token distribution [69]. For temperature $\tau > 0$, larger τ makes the distribution more uniform (wider), increasing exploration, but also increasing the likelihood of sampling a syntax-breaking token.

Definition 2.1.22 (Top-p (Nucleus) Sampling)**.** Top-p sampling restricts sampling to the smallest set of tokens whose cumulative probability is at least p, then renormalizes and samples from that set. Therefore, $p = 1$ applies no truncation.

Note (Exploration–Format Trade-off)**.** The exploration-format trade-off is a tension between exploration and format stability. Stochastic decoding can help discover correct multi-step strategies, but it also increases the probability of emitting a schema-breaking token, which can terminate a tool-use trajectory before correctness is evaluated.

Therefore, tool calling is challenging because autoregressive errors compound: small per-token error increases can cause large drops in end-to-end trajectory validity. The following two theorems, reproduced from [15], summarize this effect for sequential and compositional dependence.

Theorem 2.1.23 (Depth (sequential dependence [15]))**.** Suppose each step has a nonzero probability $\epsilon > 0$ of deviating from the correct intermediate state, and suppose the probability of "spontaneous recovery" from an incorrect state is small. Then the probability of producing the correct final state decreases rapidly as the required depth grows, because every token is conditioned on the (possibly incorrect) prefix.

Theorem 2.1.24 (Width (compositional dependence [15]))**.** Similarly, if a task requires combining n sub-computations (e.g. producing many correct entries of a matrix product), and the probability of an accidental collision (i.e., matching the verifier by guessing) is negligible, then the probability of full correctness is bounded by a term that decays approximately like $(1-\epsilon)^n$, where $\epsilon > 0$ is the per sub-computation error probability.

Therefore, extended tool use interactions are limited by a compounded risk: if the per-turn probability of emitting an executable (schema-valid) tool step is p, then a trajectory that requires N tool turns is executable with probability at most p^N. Moreover, non-zero-temperature decoding typically lowers format reliability (reducing p), so the compounded bound worsens. Therefore, long tool-call trajectories are challenging because validity constraints compound across turns.

Definition 2.1.25 (Program-Aided Language)**.** Program-Aided Language (PAL) [19] is the tool-use paradigm where the model predicts an executable action rather than a final answer directly.

Therefore, we use PAL for linear algebra, so intermediate computations run deterministically in Python/SymPy and can be verified, instead of being produced in free-form text.

Example 2.1.26 (PAL-style tool use)**.** Table 2.1 contrasts standard CoT with PAL-style tool use for linear algebra, illustrating how tool calls turn intermediate results into an executable, verifiable output.

2.2 Data Generation and Distillation

Overview. This section defines the data pipeline that is used throughout later training stages. We describe how verified synthetic linear-algebra problems are constructed via controllable generation and code execution. Then, we define the distillation and filtering terms that convert these problems into replayable trajectories and retain only those traces that pass correctness checks.

Standard Chain-of-Thought (CoT)	Tool-Integrated Reasoning (Tool Use)
Step 1: calculate the transpose of matrix A = [[3, -4], [2, 1]]. **Step 2: what is $\lVert B \rVert_F$ for B = the result from step 1?** **Thought:** I need to solve this step-by-step. First, I will calculate the transpose of Matrix A. The transpose involves swapping rows and columns. $A = \begin{bmatrix}3 & -4\\2 & 1\end{bmatrix}$ Step 1: Transposing A to get B. Row 1 of A becomes Col 1 of B: $[3, -4] \to [3, 2]$. Row 2 of A becomes Col 2 of B: $[2, 1] \to [-4, 1]$. $B = \begin{bmatrix}3 & 2\\-4 & 1\end{bmatrix}$ **Step 2: Calculating Frobenius Norm.** The formula is the square root of the sum of the absolute squares of its elements: $\sqrt{\sum \lvert b_{ij} \rvert^2}$. Calculation: $(3)^2 = 9$ $(2)^2 = 4$ $(-4)^2 = 16$ (Negative becomes positive) $(1)^2 = 1$ Summing these up: $9 + 4 + 16 + 1$. $13 + 16 + 1 = 30$. Total Sum $= 30$. Now, take the square root: $\sqrt{30}$. I know $5^2 = 25$ and $6^2 = 36$. It is halfway between 5 and 6. Let's estimate ≈ 5.48. **Answer.** 1. The transpose B is [[3, 2], [-4, 1]]. 2. The Frobenius norm is approximately 5.48.	**Step 1: calculate the transpose of matrix A = [[3, -4], [2, 1]].** **Step 2: what is $\lVert B \rVert_F$ for B = the result from step 1?** **Thought:** I need to perform matrix operations. I will use the tool to ensure the calculation is precise. **Tool call:** `matrix_transpose` **Arguments (JSON):** `{"matrix": [[3, -4], [2, 1]]}` **Tool Output.** [[3 2] [-4 1]] **Thought:** Now I have the transpose B. I need to calculate its Frobenius norm. **Tool call:** `frobenius_norm` **Arguments (JSON):** `{"matrix": [[3, 2], [-4, 1]]}` **Tool Output.** 5.477225575 **Answer.** Step 1: The transpose matrix B is: $\begin{bmatrix}3 & 2\\-4 & 1\end{bmatrix}$ Step 2: The Frobenius norm $\lVert B \rVert_F$ is 5.4772.

Table 2.1: Comparison of Standard CoT vs. tool-integrated reasoning. CoT spends tokens on explicit arithmetic and approximations (e.g. $\sqrt{30} \approx 5.48$); tool-integrated reasoning instead makes schema-valid calls (e.g. `frobenius_norm`) and uses returned values.

2.2.1 Synthetic Dataset Generation

Definition 2.2.1 (Procedural Generation)**.** Procedural generation constructs problems programmatically by sampling from controlled probability distributions (e.g. using an entropy range to tune diversity). Concretely, a generator can sample a matrix $A \in \mathbb{R}^{N \times N}$ from bounded integer ranges, then assemble a prompt whose ground truth is computed by an external tool.

Definition 2.2.2 (Verifier)**.** An executable ground truth is a deterministic function that can run and validate a solution trace end-to-end. Therefore, we accept an instance only if each intermediate tool call executes (e.g. in SymPy) and the final result matches the computed ground truth.

Definition 2.2.3 (Entropy)**.** Entropy controls how diverse the sampled numeric values are within a fixed range. Therefore, higher entropy yields more varied values, while lower entropy yields more repeated values. Consequently, we tune entropy to avoid overly repetitive data and to prevent sampled values from being excessively large. Chapter 3 calibrates this setting empirically before generating the final dataset.

Definition 2.2.4 (Uniform Distribution)**.** A (continuous) uniform distribution [6] is a model for a bounded random variable with *no preference* within an interval. Concretely, if $X \sim \mathrm{Uniform}(a, b)$, then X is equally likely to take any value between a and b. Therefore, it assigns constant density on $[a, b]$ and zero density outside:

$$p(x) = \begin{cases} \frac{1}{b-a} & \text{if } a \le x \le b, \\ 0 & \text{otherwise.} \end{cases}$$

Usage. In Chapter 3, we use uniform sampling to draw entropy budgets within specified ranges (e.g. $E_i \sim \text{Uniform}(a_i, b_i)$ for each sub-component of a generator). Therefore, the generator explores the full range of allowed budgets and avoids a fixed allocation pattern, which produces stable numerical distributions that improve training stability.

Definition 2.2.5 (Dirichlet Distribution)**.** A Dirichlet distribution [6] samples *random proportions* over K options. Concretely, it returns a vector $\pi = (\pi_1, \ldots, \pi_K)$ where $\pi_i \geq 0$ and $\sum_i \pi_i = 1$. Therefore, π can be used as weights to split a fixed budget across K parts.

The Dirichlet concentration hyperparameter $\alpha \in \mathbb{R}^K_{>0}$ controls how even or spiky the proportions are. If α is symmetric (all entries equal), then no option is preferred a priori; moreover, larger values yield more even splits, while values below 1 tend to produce more extreme splits where one part dominates.

Usage. In our implementation, Dirichlet is only used within a single component to split that component's entropy between at most two constituent parts. Consequently, we use $K = 2$ with $\alpha = c \cdot (1, 1)$ for a tunable concentration c (by default c is 1.0).

Example 2.2.6 (Internal Dirichlet split)**.** Suppose a component has entropy budget $E = 1.0$ and needs to split it between two internal parts (e.g. sampling a matrix versus sampling a vector). We draw $\pi \sim \text{Dir}(c \cdot (1, 1))$ which might yield $\pi = (0.7, 0.3)$. Therefore, the component allocates $0.7E$ to the first part and $0.3E$ to the second part, while the total remains fixed ($0.7E + 0.3E = E$).

2.2.2 Knowledge Distillation and Rejection Sampling

Definition 2.2.7 (Knowledge Distillation)**.** Knowledge Distillation (KD) is effectively a form of knowledge transfer between a large teacher model to a small student model [25]. The primary forms of knowledge distillation include offline distillation, online distillation, and self-distillation [20].

Definition 2.2.8 (Offline Distillation)**.** Offline distillation is a two-phase process in which a pre-trained teacher model transfers its knowledge to a student model while the teacher's weights remain frozen [25]. This is the distillation scheme used throughout this thesis.

Other schemes. Online distillation updates student and teacher-like peers simultaneously (e.g. mutual-learning variants) [87], while self-distillation uses the same architecture as its own teacher across training stages [17].

Note (Why Offline Distillation)**.** Offline distillation is commonly used because it is simple, decouples generation from training, and allows the teacher to be deployed independently. In our setting, this separation is particularly useful: we can mine and filter tool-use traces with a large teacher, then train the student on the resulting data using reduced compute.

Definition 2.2.9 (Chain of Thought Traces [77])**.** Distillation data can include intermediate reasoning steps alongside final predictions. In this thesis, these steps are included in the tool-use trace, so the student is trained to reproduce the same reasoning structure, and the traces are replayable rather than treated as free-form text.

Definition 2.2.10 (Rejection Sampling)**.** Rejection sampling is a filtering procedure that repeatedly samples candidate trajectories and retains only those that pass a validation test. The validation test is end-to-end executability: a trace is accepted only if it can be parsed under the tool-call schema, replayed without execution errors, and verified against the ground truth.

2.3 Fine-Tuning and Alignment

Overview. This section introduces the post-training objectives: SFT with Low-Rank Adaptation (LoRA) [27] and GRPO.

Definition 2.3.1 (Pre-Training)**.** Pre-training optimizes a Transformer language model for next-token prediction on large, general dataset, yielding broad language competence but not guaranteeing instruction-following or valid tool-call syntax.

Because of this, we apply LoRA-based SFT to stabilize the tool-use interface. Then, GRPO further improves multi-turn correctness using reinforcement learning with verifiable rewards.

Definition 2.3.2 (Full-Parameter Fine-Tuning)**.** Full-parameter fine-tuning adapts a pre-trained language model to a target task by updating all parameters on task-specific data.

Formally, given an autoregressive model $P_\Phi(y|x)$ initialized at Φ_0 and a dataset $\mathcal{Z} = \{(x_i, y_i)\}_{i=1}^N$, fine-tuning solves

$$\Phi^* = \arg\max_{\Phi} \sum_{(x,y)\in\mathcal{Z}} \sum_{t=1}^{|y|} \log P_\Phi(y_t \mid x, y_{<t}).$$

However, because all weights are updated, compute and memory cost scales with $|\Phi|$; moreover, large updates can increase the risk of catastrophic forgetting. Therefore, we use parameter-efficient updates (LoRA) rather than full-parameter training.

Definition 2.3.3 (Low-Rank Adaptation (LoRA) [27])**.** Low-Rank Adaptation (LoRA) is a parameter-efficient fine-tuning method that freezes the pre-trained weights and learns a low-rank update for selected weight matrices.

To be concrete, for a dense weight matrix $W_0 \in \mathbb{R}^{d\times k}$, LoRA parameterizes the adapted weight as

$$W = W_0 + BA,$$

where $B \in \mathbb{R}^{d\times r}$ and $A \in \mathbb{R}^{r\times k}$ with $r \ll \min(d, k)$. Consequently, the number of trainable parameters scales as $r(d + k)$ rather than dk.

Moreover, training updates only the LoRA matrices $\{A, B\}$ under the standard language-model loss, while the base parameters Φ_0 remain fixed.

Therefore, LoRA reduces memory and optimizer state, making SFT feasible on limited hardware; in this thesis, we apply it to projection matrices commonly used in attention.

2.3.1 Supervised Fine-Tuning (SFT)

Definition 2.3.4 (Supervised Fine-Tuning (SFT))**.** Given a dataset $\mathcal{D}_{\text{SFT}} \triangleq \{(q_i, c_i)\}_{i=1}^{|\mathcal{D}_{\text{SFT}}|}$ where each sample (q_i, c_i) pairs a question q_i with a reference completion c_i (a multi-turn trajectory containing reasoning text and tool calls, terminating in a final answer), supervised fine-tuning updates the policy π_θ by minimizing the negative log-likelihood:

$$\mathcal{L}_{\text{SFT}}(\theta) \triangleq -\mathbb{E}_{(q,c)\sim\mathcal{D}_{\text{SFT}}}\left[\log \pi_\theta(c \mid q)\right],$$

where $\pi_\theta(c \mid q)$ denotes the probability assigned by the model π_θ to the trajectory c conditioned on the question q. Therefore, this objective encourages the model to imitate the supervised demonstrations by maximizing the likelihood of the reference completions.

Suppose that each sample is equally likely to be chosen, then the above expectation objective ($\mathbb{E}$) can be calculated by averaging the negative log-likelihood loss across all training examples in the dataset:

$$\mathcal{L}_{\text{SFT}}(\theta) = -\frac{1}{|\mathcal{D}_{\text{SFT}}|}\sum_{i=1}^{|\mathcal{D}_{\text{SFT}}|} \log \pi_\theta(c_i \mid q_i)$$

2.3.2 Reinforcement Learning (GSPO)

Algorithm 2.3.5 (GSPO – GRPO variant)**.** A language model policy π_θ assigns probabilities to complete trajectories τ given a prompt. Therefore, in contrast to SFT (which fits a fixed reference completion from a supervised dataset), reinforcement learning updates θ to increase the probability of high-reward sampled trajectories.

Figure 2.1 summarizes the critic-free, group-relative loop used in this thesis. Because tool-use rewards are trajectory-wide over many turns, GRPO [64] can be high-variance and unstable; therefore, we use Group Sequence Policy Optimization's [89] (GSPO) sequence-level ratio and Proximal Policy Optimization (PPO) [61] style clipping (the objective G.8 in Figure 2.1) to stabilize long-horizon updates. Consequently, we decompose the update into the following steps:

1. **Generating completions (`G.1`–`G.3`).** Firstly, for each prompt q we sample a group of G trajectories $\{\tau_1, \ldots, \tau_G\}$ from the rollout policy $\pi_{\theta_{\text{old}}}$, where each trajectory corresponds to a token completion $o_i = (o_{i,1}, \ldots, o_{i,|o_i|})$.
2. **Computing the advantage (`G.4`–`G.5`).** Secondly, we assign scalar rewards $r_i = R(\tau_i)$ and compute a group-relative advantage by normalizing within the group:

   $$\hat{A}_i = \frac{r_i - \text{mean}(\mathbf{r})}{\text{std}(\mathbf{r})}, \quad \mathbf{r} = (r_1, \ldots, r_G),$$

   then use $\hat{A}_i$ as a trajectory-level weight for the policy update. Moreover, implementations may disable std($\mathbf{r}$) scaling or compute the standard deviation at the batch level to reduce difficulty-dependent bias (nonetheless we preserve it).

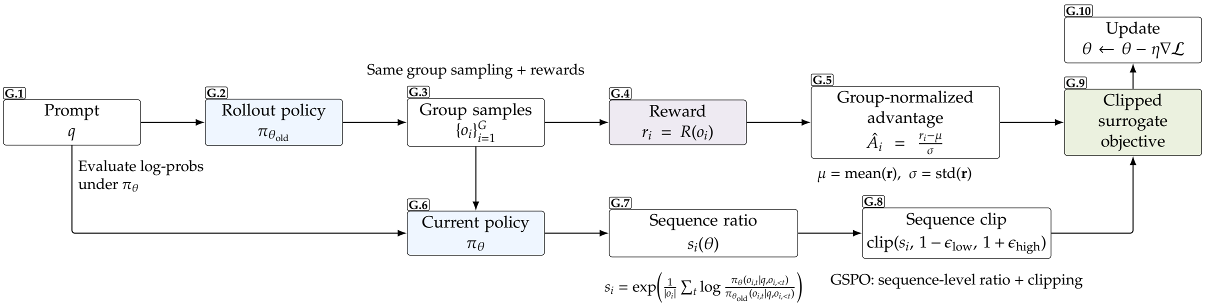

G.1
Prompt
q
G.2
Rollout policy
$\pi_{\theta_{\text{old}}}$
Same group sampling + rewards
G.3
Group samples
$\{o_i\}_{i=1}^{G}$
G.4
Reward
$r_i = R(o_i)$
G.5
Group-normalized advantage
$\hat{A}_i = \frac{r_i - \mu}{\sigma}$
$\mu = \text{mean}(\mathbf{r}),\ \sigma = \text{std}(\mathbf{r})$
G.9
Clipped surrogate objective
G.10
Update
$\theta \leftarrow \theta - \eta \nabla \mathcal{L}$
Evaluate log-probs under π_θ
G.6
Current policy
π_θ
G.7
Sequence ratio
$s_i(\theta)$
$s_i = \exp\left(\frac{1}{|o_i|} \sum_t \log \frac{\pi_\theta(o_{i,t}|q,o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t}|q,o_{i,<t})}\right)$
G.8
Sequence clip
$\text{clip}(s_i, 1-\epsilon_{\text{low}}, 1+\epsilon_{\text{high}})$
GSPO: sequence-level ratio + clipping

Figure 2.1: GSPO training loop. `G.1--G.5` sample and score a group under the old policy, `G.6--G.8` compute the sequence-level ratio and clipping used by GSPO, and `G.9--G.10` apply the clipped surrogate update.

3. **Computing the sequence ratio (`G.6–G.7`).** Thirdly, we evaluate the sampled group under the current policy and form a length-normalized sequence likelihood ratio [89]:
$$s_i(\theta) = \exp\left(\frac{1}{|o_i|} \sum_{t=1}^{|o_i|} \log \frac{\pi_\theta(o_{i,t} \mid q, o_{i,<t})}{\pi_{\theta_{\text{old}}}(o_{i,t} \mid q, o_{i,<t})} \right).$$

4. **Clipping (`G.8`).** Fourthly, we clip the sequence ratio to prevent large updates:
$$\bar{s}_i(\theta) = \text{clip}\Big(s_i(\theta),\ 1 - \epsilon_{\text{low}},\ 1 + \epsilon_{\text{high}}\Big).$$

5. **Computing the loss and update (`G.9–G.10`).** Finally, we optimize a PPO-style clipped surrogate objective at the sequence level [60, 89]:
$$\mathcal{L}_{\text{GSPO}}(\theta) = -\frac{1}{G} \sum_{i=1}^{G} \min\Big(s_i(\theta)\hat{A}_i,\ \bar{s}_i(\theta)\hat{A}_i\Big).$$

2.4 Evaluation Methodology

The following definitions are referenced when reporting the results in Chapters 5 and 6.

Definition 2.4.1 (Verifier-Based Evaluation)**.** Verifier-based evaluation deterministically executes the model's proposed trajectory: (1) parse and validate the structured output under the format contract; (2) execute tool calls in order; and (3), compare the final answer against ground truth produced by the same verifier. Therefore, a trajectory can fail either because it is not executable (format, or tool error) or because it is incorrect.

Definition 2.4.2 (Optimal Trajectory)**.** Optimal trajectory measures whether a rollout achieves the maximum composite success reward (correctness, format validity, tool success, and no extra tool calls); it is reported as the mean of this binary indicator over evaluation tasks.

Definition 2.4.3 (Correctness)**.** Correctness measures whether the verifier confirms the final answer matches the ground truth for the task.

Definition 2.4.4 (Format Validity)**.** Format validity measures whether the model output satisfies the required serialization schema and is parseable into a well-formed trajectory (e.g. well-formed tags and well-formed JSON arguments for tool calls). Moreover, this check is applied before any tool execution.

Definition 2.4.5 (Tool Success)**.** Tool success measures whether each parsed tool call executes without runtime error in the external environment.

Definition 2.4.6 (Trajectory Validity)**.** Trajectory validity measures whether the full multi-turn solution is correctly formed and follows the expected tool-use format, meaning that the generated trajectory contains no structural errors.

Definition 2.4.7 (Training Stability)**.** Training stability means that optimization does not diverge: losses remain finite, gradient norms stay bounded, and update statistics (e.g. importance ratios or sequence ratios prior to clipping) stay within a controlled range over time. Therefore, stability is assessed through these signals and by checking that constraint metrics (format validity, tool success) do not collapse abruptly.

Note (Compounding risk)**.** Because a tool-using trajectory is a chain of dependent events, constraint satisfaction compounds multiplicatively. For example, if per-turn format validity is p_t and a task requires N tool turns, then an upper bound on valid trajectories is p_t^N.

Moreover, this compounding is amplified when training with exploration parameters (temperature/top-p 1.0): if the policy's next-token distribution is high-entropy (as in a pre-trained base model before SFT), stochastic decoding is more likely to drift away from the small set of format control tokens, and the per-turn failure probability compounds across the trajectory.

Note (Evaluation approach)**.** Therefore, when reporting headline numbers, our approach fixes the evaluation split and the decoding configuration, so results are comparable across phases. This means that evaluation uses greedy decoding (argmax at each step), while training uses stochastic sampling with temperature=1.0 and top-p=1.0.

2.5 Summary

This chapter defined the shared vocabulary needed to follow the end-to-end pipeline. Firstly, it framed decoder-only Transformers as autoregressive sequence models and motivated tool use as generating executable, multi-turn trajectories rather than single-shot answers.

Moreover, it characterized tokenization, chat templates, structured outputs, and stop conditions as concrete constraints on what a model can emit. Therefore, it motivates special tokens and strict serialization as practical requirements for reliable tool use.

Finally, it defined the data terms that bridge verified synthetic problems to distilled and rejection-filtered traces, introduced the post-training objectives used later (SFT with LoRA and GSPO as a GRPO variant), and specified verifier-based evaluation metrics under a fixed decoding protocol, so results remain comparable across phases.

Chapter 3

Dataset Generation

3.1 Related Work

Generation Paradigms. Our goal is to build a linear algebra tool dataset that allows deterministic verification. In this context, synthetic data pipelines are typically divided into two families [40]: (i) *bootstrapping* [82] methods that rewrite existing problems into more rephrased variants, and (ii) *procedural generators* [58] that sample fresh instances from controlled distributions and compute ground truth by code execution. We mainly use procedural generation, but also apply light rephrasing to increase data diversity.

Furthermore, several recent works argue that synthetic data is most useful for post-training when each instance is uniquely verifiable, especially when the same data will later be used in reinforcement learning [2]. Consequently, our pipeline ensures this constraint by following a generate → execute → verify → accept loop, where input data (e.g. a sampled matrix $A \in \mathbb{R}^{N \times M}$) is paired with a unique final answer.

Dataset Formats. Math datasets span multiple formats and subdomains, from short word problems and formal proofs to visual questions. Common ways to compile such datasets are by curating public forums and open educational resources [67, 3, 49, 4].

However, most math datasets are single-turn question–answer pairs, whereas our setting requires interactive tool use. Because of this, conventional data collection makes it challenging to obtain the multi-step tool-interaction traces required by the ReAct paradigm [81]. Therefore, synthetic generation is optimal in this scenario because it scales multi-step trajectories, controls step count, and deterministically verifies results. Consequently, it supports step-based curriculum training that stabilizes RL [22, 78].

Tool Learning. Following the tool-learning survey in [76], we treat a *tool* as an external program the model can call by generating structured arguments. Therefore, tool use is typically framed as a loop: plan, select a tool, call it, and use the result [53].

Tool calling is studied in single-shot and iterative settings. In this setting, a one-step workflow is often described as *planning without feedback*, while a multi-step workflow is *planning with feedback*, because each tool output becomes new context that constrains the next decision [74]. Our generation pipeline, illustrated in Figure 3.1, targets planning with feedback by generating problems that require repeated tool interactions.

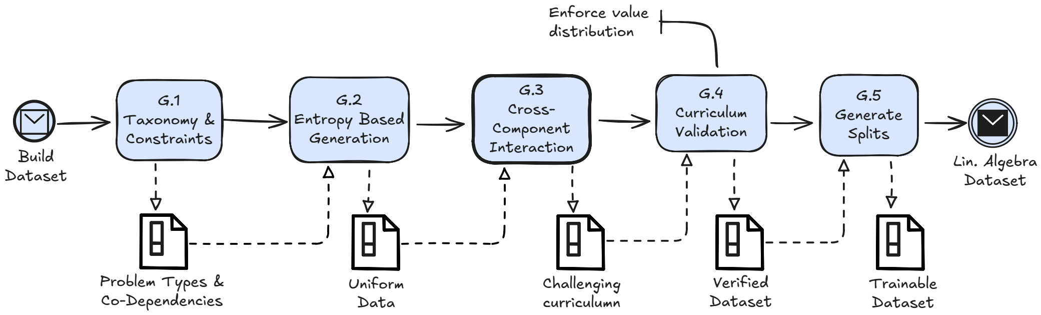

Enforce value distribution
Build Dataset
G.1 Taxonomy & Constraints
G.2 Entropy Based Generation
G.3 Cross-Component Interaction
G.4 Curriculum Validation
G.5 Generate Splits
Lin. Algebra Dataset
Problem Types & Co-Dependencies
Uniform Data
Challenging curriculum
Verified Dataset
Trainable Dataset

Figure 3.1: Dataset generation workflow schematic (G.1–G.5). Defines the task taxonomy and constraints, samples values under entropy control, composes multi-step tool chains, verifies executability and bounds, and produces fixed-seed train/validation/test splits.

3.2 Overview

Motivation. High-quality linear algebra tool-use data with deterministic verification is scarce in publicly available sources [53]. Therefore, open sources pose two practical constraints: they rarely provide a uniform, tool-executable format with reliable correctness, and they are difficult to curate at the scale required for modern post-training.

One limitation is that open-source math data is heterogeneous, so collecting it from forums or textbooks tends to produce inconsistent solution structure and weak guarantees of correctness. Therefore, even widely used benchmarks such as GSM8k [8] and MATH [23] do not enforce a single executable format that can be replayed for deterministic verification. Because of this, they are an awkward fit for tool-use supervision, where intermediate steps must follow a strict format contract.

A second limitation is that curation is costly and hard to verify end-to-end, and early math datasets were too small for modern post-training [29, 39]. For instance, aggregated datasets are still small (GSM8K at 8k; MATH at 12k) [83]. Consequently, because full fine-tuning benefits from > 100k examples [31], synthetic generation is the practical way to scale data while preserving a uniform, verifiable format for full fine-tuning and LoRA, with peak LoRA performance reachable at around 8k examples [7].

Solution. Therefore, we use procedural generation to produce linear-algebra tool-use examples in a uniform, executable schema with a single verified solution. Consequently, the final dataset contains 10,560 examples spanning 13 problem types, which is sufficient for LoRA-based SFT and LoRA-based GRPO under the dataset-size guidance of [63]. For concrete samples from the released dataset, see Appendix C.

3.2.1 Taxonomy & Constraints

G.1 Taxonomy & Constraints. This phase defines the task taxonomy, tool interface, and validity constraints for generated samples. One key constraint is the format contract that each dataset record must satisfy (see Figure 3.2), which standardizes tags and tool-call structure for deterministic execution and later training.

Question: Given matrix A = [[-120, -2], [-22, -35]], find $\|A\|_F$.

Final Answer: 126.94

Difficulty: Easy

Topic: Linear Algebra, Frobenius Norm, Single Turn.

Template: <think>The user wants me to find the Frobenius norm of matrix A, which is given as $[[-120, -2], [-22, -35]]$ [...]. I shouldn't do the calculations manually. The user provided a tool called frobenius_norm that does exactly this. So I should use that tool.</think>
<tool_call>
frobenius_norm([[-120, -2], [-22, -35]])
</tool_call>
<tool_response>126.94</tool_response>
<think>Now I need to provide the final answer.</think>
<answer>126.94</answer>

Figure 3.2: Example synthetic dataset record illustrating only the tool-execution contract (structured tags and tool calls); the shown solution is included only to demonstrate tag usage.

Interface Contract. The interface contract specifies the dataset format expected by later training phases. Concretely, it instantiates the *format contract* and *verifier* vocabulary from Chapter 2: each example includes a natural-language query, a sequence of tool-call inputs, and a ground-truth final answer. Figure 3.2 illustrates this format, including a tool-grounded solution and the required special tags. Therefore, we accept an example only if executing the full tool-call sequence reproduces the stored final answer exactly.

Tools and Reproducibility. Furthermore, because tool use is only as effective as the function signatures it targets, we explicitly list the supported operations and their argument/return formats. Consequently, Table 3.1 enumerates the atomic tool operations that the generator composes into 1-step, 2-step, and 3-step problems. Finally, we fix both the generator seed and the split seed to make data generation and train/validation/test partitioning reproducible.

Curriculum Primitives: Atomic Operations

Atomic operations. Firstly, we define a small set of atomic operations that make difficulty controllable and facilitate validation for both single-step and composed instances. Because of this, each primitive is paired with a deterministic tool implementation (e.g. rank, determinant, cofactor), so every sub-result can be executed and checked. Consequently, we define difficulty by restricting traces to $N \in (1, 2, 3)$ tool calls while keeping the operator vocabulary fixed.

Compositionality. Because of this fixed primitive set, we generate problems bottom-up by composing atomic linear algebra operations into longer tool-call chains. This compositional approach increases diversity while keeping every intermediate step executable.

Primitive list. Table 3.1 defines the generator's core operations: transpose, trace, determinant, frobenius norm, rank, and cofactor. For precise definitions and simple examples, see Appendix A.

`Function Name`	Description	Input Constraint	Return
`matrix_transpose`	Computes the transpose.	Any Matrix	Matrix
`matrix_cofactor`	Calculates the matrix of cofactors.	Square Matrix	Matrix
`determinant`	Computes the determinant of a matrix.	Square Matrix	Float
`frobenius_norm`	Calculates the Frobenius norm.	Any Matrix	Float
`matrix_rank`	Determines the rank of a matrix.	Any Matrix	Integer
`matrix_trace`	Calculates the trace of a matrix.	Square Matrix	Float

Table 3.1: The Linear Algebra Tools Available to the Agent. Complete description of operations is available in Appendix A.

Problem classification. Moreover, we define 13 problem types as operator compositions over the tool vocabulary (6 single-step, 4 two-step, and 3 three-step chains). Therefore, the full taxonomy is reported in Appendix B. A representative record is shown below, while additional samples appear in Appendix C.

Problem type. `two_transpose_frobenius`.
Query.

```
Step 1: find the transpose of matrix A = [[-35, -6], [43, 20]].
Step 2: find the Frobenius norm of matrix B = the result from step 1.
```

Ground truth. 59.25.
Stepwise ground truth.

```
[
  {"matrix_transpose": [[-35, 43], [-6, 20]]},
  {"frobenius_norm": 59.25}
]
```

3.3 Optimizations/Analysis

3.3.1 Design patterns

Core patterns. We use a small set of design patterns to structure the dataset generator, drawing primarily on the Gang of Four catalog [18] and Python-focused guidance from [52]. Therefore, we use these patterns to make the generator easier to maintain, extend, and understand as the curriculum grows:

- **Factory + Registry.** Encapsulates task creation and centralizes problem generation; consequently, new problem types can be added without editing the problem generation code.
- **Template Method.** Enforces the generation lifecycle (generate → render → verify); therefore, every task shares the same rendering and validation mechanism via `TemplateEngine`.
- **Composite + Strategy.** Builds 2-step and 3-step problems by composing atomic components under a given dependence policy (i.e. sequential dependence); because of this, it is simple to change composition rules without extensive local modifications.

- **Adapter + Builder.** Adapts single-step generators into composable components and assembles a final artefact from intermediate tool outputs; consequently, multi-step queries are composed based on the specific atomic types.

Figure 3.3 illustrates the dataset generation pipeline and highlights each design pattern. Firstly, the orchestration layer uses the CLI entrypoint to select tasks via the factory registry and instantiate them through a task factory. Then, the generation layer branches into single-step and multi-step generation, where multi-step problems are built by composing components under a sequential composition policy and assembling intermediate outputs into a single query. Finally, both paths converge in the export layer, where rendering, verification, and dataset export produce user queries, tool-call traces, and splits.

3.3.2 Entropy Based Generation

G.2 Entropy Based Generation. Procedural generation raises an important question: how do we avoid trivial or excessively large numbers while keeping the dataset diverse and solvable? Because linear algebra operations can amplify values nonlinearly

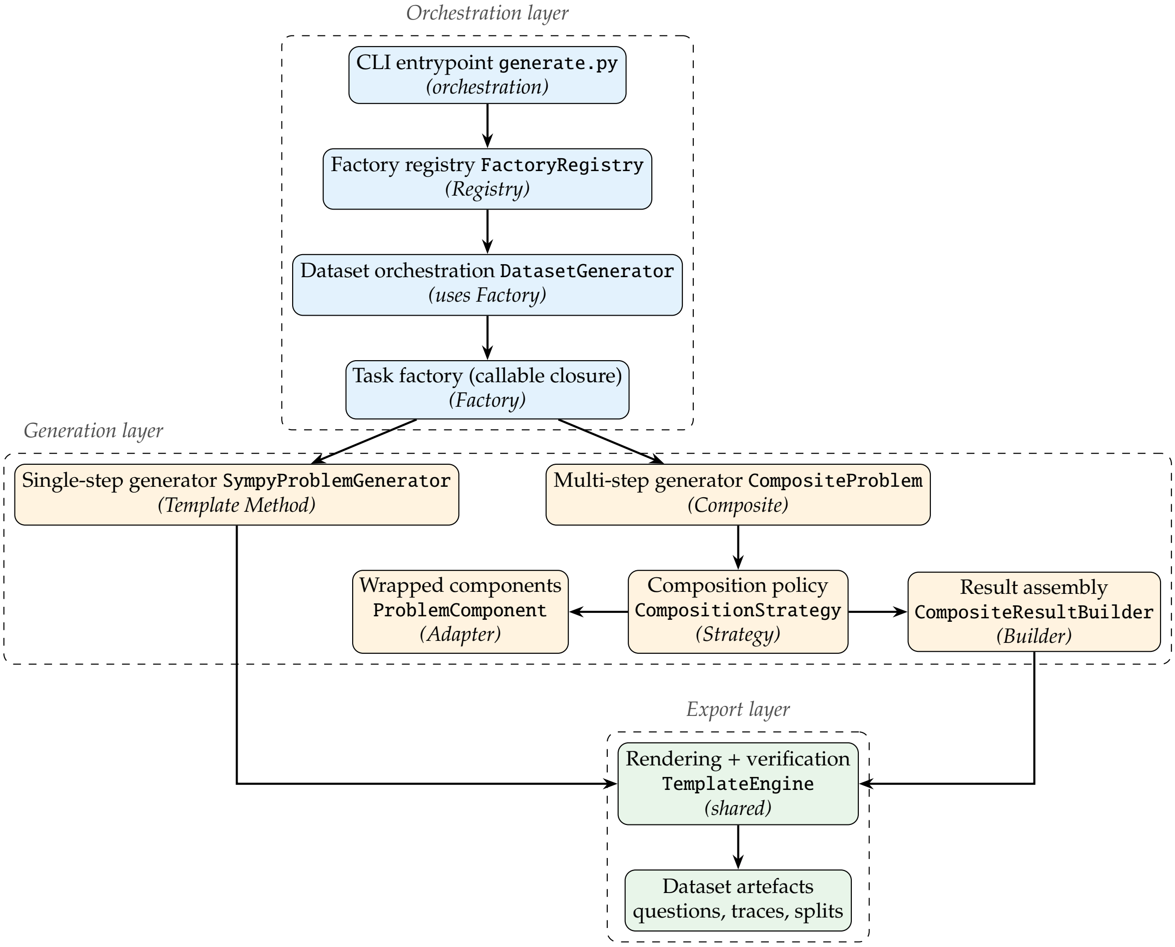


Figure 3.3: Conceptual workflow of the dataset generation pipeline and the locations of the design patterns discussed in Section 3.3.1.

(e.g. determinants), naive sampling can also cause value explosion and unstable distributions. Therefore, we parameterize sampling with an entropy range that stabilizes the distribution of generated values. Algorithm 3.1 summarizes how this budget is allocated across components for multi-step problems. For single-step problems see Appendix D.

Per-component budgets. We adapt an entropy-controlled sampling mechanism, inspired by [58], to keep generated values within stable bounds while still spanning easy-to-hard instances. Therefore, rather than splitting a single fixed entropy budget via a normalized partition (e.g. in a way that the constituents sum to 1), each component $i \in \{1, \dots, K\}$ is assigned its own entropy range $[a_i, b_i]$. For each instance, we independently sample a per-component budget E_i from this range – typically $E_i \sim \text{Uniform}(a_i, b_i)$ – and execute component i with entropy budget E_i. As a result, the overall entropy of the composite instance is $E = \sum_{i=1}^{K} E_i$, so variability arises from per-component sampling rather than from a single normalized split; this reduces the risk of concentrating difficulty in the same component across instances.

Example 3.3.1 (Entropy-controlled sampling)**.** A matrix-based generator first samples a shape (n, m) from a bounded set (e.g. $n, m \in \{2, 3\}$). Next, it samples an entropy value E from a configured range, typically $E \sim \text{Uniform}(a, b)$, and uses E to control the magnitude of sampled integers: entries are drawn by sampling uniformly from a symmetric integer interval whose width scales on the order of 10^E, while enforcing a minimum absolute value (rejecting samples with $|x| <$ `min_element_abs`). If additional structural constraints are required – e.g. the matrix must be invertible – the generator applies rejection sampling, resampling until the constraint holds (such as $\det(A) \neq 0$). Thus, larger E tends to produce larger and more diverse numerical instances while

```
procedure GenerateQuestionComposite(problemType, components[1..k])
    baseCtx ← CompositionContext(entropy=0)
    baseCtx.constraints ← {}
    results ← [ ]
    for i ← 1 to k do
        E ← SampleEntropy(components[i].entropyRange)
        if components[i].entropyWeight = 0 then
            E ← 0                                              ▷ pure transform
        end if
        ctx ← CompositionContext(entropy=E)
        ctx.constraints ← Copy(baseCtx.constraints)            ▷ preserve constraints
        ctx.componentResults ← results                         ▷ enforce step deps.
        result ← RunComponent(components[i], ctx)
        results.append(result)
        baseCtx ← Merge(baseCtx, ctx)                          ▷ ground truth + constraints
    end for
    finalTemplate ← BuildCompositeTemplate(results)
    return (finalTemplate, baseCtx.stepwiseGroundTruths)
end procedure
```

Algorithm 3.1. Entropy-controlled composite (multi-step) generation. Allocates per-component entropy budget, allocates step dependencies, and records the user query.

keeping matrix dimensions within fixed bounds.

Two-level entropy allocation. Figure 3.4 illustrates how random numbers are generated on a per-component basis. Firstly, we use uniform sampling to draw an *absolute* entropy budget from a configured range and assign it to a single component (e.g. $E_i \sim \text{Uniform}(a_i, b_i)$). On the other hand, some generators instantiate more than one entity and must allocate that component budget internally rather than resampling unrelated scales. Therefore, we use a Dirichlet distribution [1] to sample *relative* proportions and split the budget across entities; for example, a linear system generator requires both a matrix $A \in \mathbb{R}^{n\times n}$ and a vector $b \in \mathbb{R}^n$, so we draw $(p_A, p_b) \sim \text{Dirichlet}(\mathbf{1})$ and set $(E_A, E_b) = (p_A E_i, p_b E_i)$. Consequently, uniform sampling controls global difficulty at the component level while Dirichlet controls within-component balance, and this way the generator avoids both value explosion and trivial cases where one entity dominates the instance.

G.3 Cross-Component Interaction. Lastly, multi-step problems are created by sequentially composing atomic operations rather than by hardcoding complex templates.

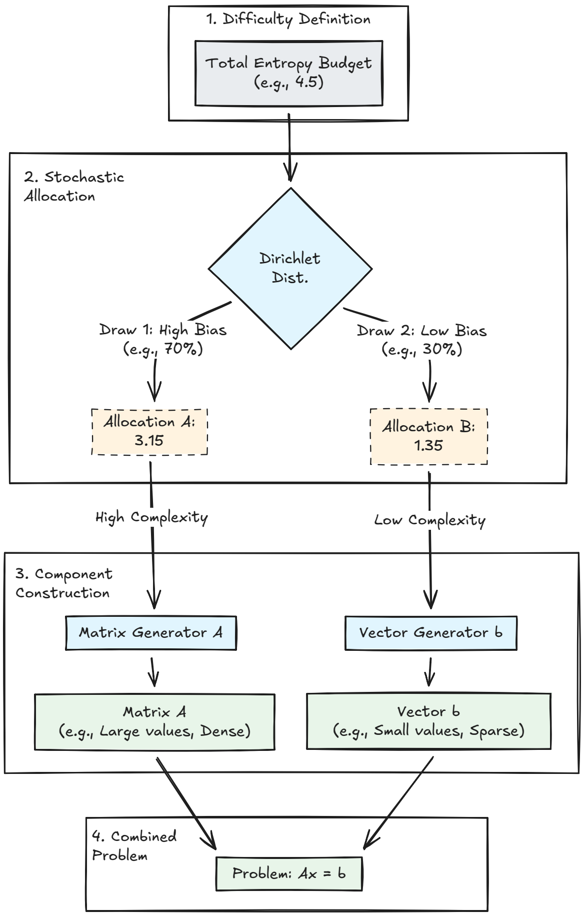


Figure 3.4: Entropy-controlled sampling strategy. Varies numeric scale across components to ensure values within a predefined range.

Because of this, 2-step and 3-step questions explicitly consume the output of prior steps during replay, and the dataset can record step dependencies alongside the tool-call inputs. Figure E.1 from Appendix E illustrates this composition mechanism.

3.4 Experiments

3.4.1 Curriculum Validation

G.4 Curriculum Validation. This phase enforces the generator's guarantees about distribution stability by executing candidate tool-call traces, rejecting failed traces, and calibrating entropy ranges to stay within configured numeric bounds.

Verification. Each candidate instance is validated before it is commited in the dataset. Therefore, we execute the tool-call sequence, and we reject an instance if any step fails to execute or if the final result disagrees with the stored ground truth. This is the generation-stage analogue of verifier-based evaluation from Chapter 2. Figure D.1 summarizes this "generate → execute → verify → accept" loop.

Example 3.4.1 (Verification Failure Example)**.** Consider the composite task `Transpose` → `Det.`, which first constructs $A^\top$ and then computes $\det(A^\top)$. Firstly, the determinant is only defined for square matrices. Therefore, if a component samples a non-square matrix such as:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \in \mathbb{R}^{2\times 3},$$

then $A^\top \in \mathbb{R}^{3\times 2}$, and consequently the `Det.` tool call is undefined and will fail during execution.

The composite generator initializes each component with constraints derived from downstream requirements. Because the downstream `Det.` step requires a square input, composite initialization passes a flag (i.e. `requires_square=true`) into `Transpose`, constraining it to sample $A \in \mathbb{R}^{n\times n}$ so that $A^\top$ remains square.

Concretely, `Transpose` overrides the abstract `GenerateMathContent` method from the base component class; it consumes the propagated constraints and adjusts sampling accordingly. As shown in Algorithm 3.2, `Transpose.GenerateMathContent` retries until the square-matrix requirement is satisfied.

```
procedure TRANSPOSE.GENERATEMATHCONTENT(entropyBounds, constraints)
    assume constraints.requires_square = true          ▷ set by composite init
    repeat
        $A \leftarrow$ SampleMatrix(entropyBounds, constraints)
        $ok \leftarrow$ IsSquare($A$)                  ▷ shape verifier
    until ok
    return $A$
end procedure
```

Algorithm 3.2. Verification failure for `Transpose` → `Det.` and the resampling loop used to enforce the square-matrix contract.

Nonlinear scaling. Defining the complexity of linear algebra problems is non-trivial; analytical bounds are often insufficient because operations like determinants and matrix multiplications exhibit non-linear value scaling. A matrix with small integer elements can yield an excessively large determinant, and therefore naive sampling can break the verifier contract by producing unstable values. Therefore, we use a Monte Carlo calibration phase to tune the sampler's entropy ranges, so outputs stay within the bounds defined in Table 3.2.

Calibration loop. To apply these bounds, we implement a Monte Carlo calibration loop (Figure 3.5). Before the final dataset is generated, the system performs an offline grid search over the *entropy* parameter space (in the Chapter 2 sense of controlling numeric diversity under fixed bounds). For every candidate entropy value (e.g. $E = 1.2$), the system generates $N = 8,000$ synthetic samples and profiles their output distributions. Therefore, if an entropy value produces outputs outside the declared bounds, we drop that entropy setting and keep only settings that satisfy the constraints.

Bounded diversity. Consequently, we can maximize dataset diversity while strictly adhering to safety bounds. These calibrated entropy settings show that the entropy-controlled generator is effective in practice, which makes the downstream results in Chapters 4–6 more reliable, leading to fewer out-of-range failures and stable output distributions.

Compliance. Therefore, calibration provides two complementary artifacts: the loop description (Figure 3.5) and the empirical min/max compliance summary (Table 3.2). We log these observed ranges during calibration and generation to confirm that the chosen entropy settings stay within the declared bounds. In practice, we pick the highest-entropy setting that still satisfies the bounds, so we keep as much numeric diversity as possible without violating the constraints.

Task Type	**Steps**	**Optimized Entropy**	**Target Range**	**Observed Min**	**Observed Max**	**Samples**
Determinant	1	0.9	$[-500, 500]$	−417.00	410.00	8000
Matrix Trace	1	2.1	$[-200, 200]$	−164.00	166.00	8000
Frobenius Norm	1	2.6	$[0, 600]$	18.57	460.79	8000
Matrix Rank	1	2.5	$[1, 3]$	1.00	3.00	8000
Transpose	1	3.2	$[-800, 800]$	−790.00	787.00	8000
Cofactor	1	1.6	$[-800, 800]$	−735.00	779.00	8000
Transpose → Det.	2	$[0.7, 0.0]$	$[-400, 400]$	−216.00	224.00	8000
Cofactor → Trace	2	$[1.4, 0.0]$	$[-800, 800]$	−469.00	604.00	8000
Cofactor → Rank	2	$[1.5, 0.0]$	$[-800, 800]$	−480.00	480.00	8000
Transpose → Frob.	2	$[2.8, 0.0]$	$[-800, 800]$	−315.00	768.88	8000
Transp. → Cof. → Rank	3	$[3.2, 0.0, 0.0]$	$[-800, 800]$	−790.00	791.00	8000
Cof. → Transp. → Trace	3	$[2.9, 0.0, 0.0]$	$[-800, 800]$	−765.00	774.00	8000
Transp. → Cof. → Frob.	3	$[2.9, 0.0, 0.0]$	$[-800, 800]$	−395.00	699.58	8000

Table 3.2: Entropy Calibration Results. The table displays the maximum entropy settings found via grid search that satisfy the target numerical bounds, together with observed output ranges across 8,000 Monte Carlo samples per task. The configuration file is available here.

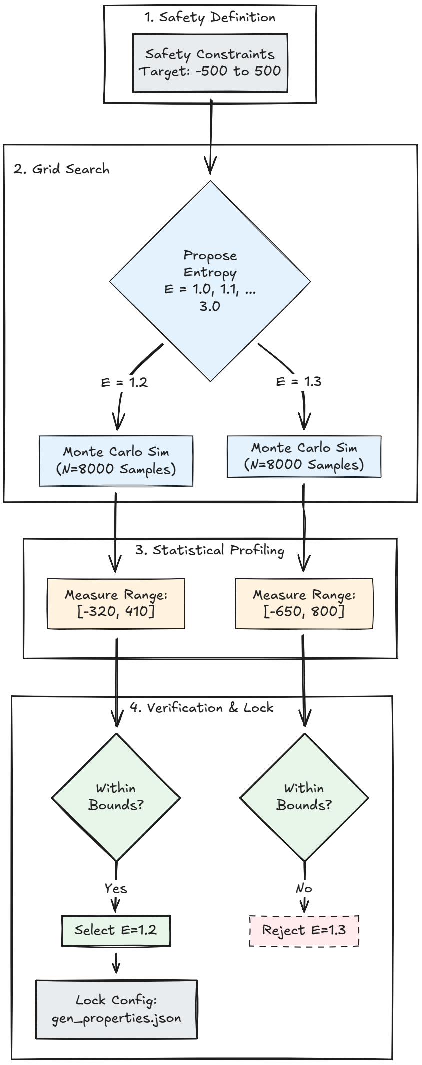


Figure 3.5: Entropy calibration loop. Offline grid search selects settings that keep outputs within target numeric bounds across Monte Carlo samples.

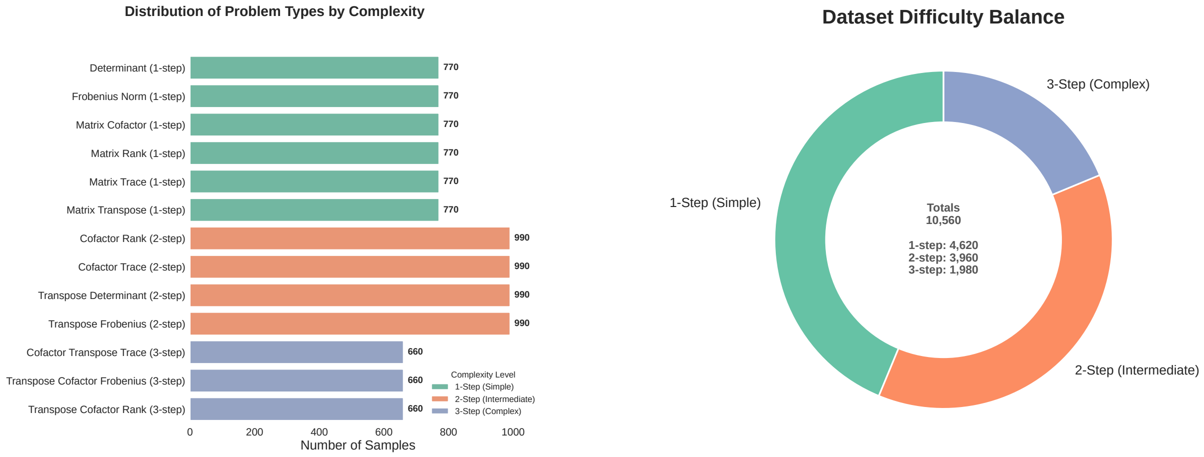
3.4.2 Generate Splits

G.5 Generate Splits. For reproducibility, we fix the dataset schema and difficulty definition, then generate stratified train/validation/test splits using a fixed seed.

Schema and difficulty. Difficulty is defined as the number of tool calls (1, 2, or 3), where matrices and scalars are serialized using JSON. This design keeps the interface parseable and replayable under a unified format contract, as defined in Figure 3.2. Moreover, because format validity compounds across a multi-turn tool-use trajectory, limiting difficulty to $N \in \{1, 2, 3\}$ keeps the executable-trace requirement tractable for later phases.

Limitations. Nonetheless, the dataset is intentionally constrained: it covers only the linear algebra operators implemented in the tool library and restricts numeric ranges to keep instances stable and replayable. Consequently, the results primarily measure the model's ability to plan and execute tool-call trajectories (and to preserve syntactic validity across turns), while reducing errors caused by excessively large arithmetic that could otherwise account for most failures.

Dataset composition. Figure 3.6 shows the dataset (spanning 13 problem types) split into 1-step, 2-step, and 3-step difficulty tiers, defined by the number of tool calls required to solve each problem. We fixed counts for each category (1/2/3 tool calls) and run generation under a fixed seed until those thresholds are met. Finally, we split the dataset into train/validation/test (80/10/10) using a fixed seed and stratify by the `difficulty` column (i.e., # tool calls), so each split preserves the same tier proportions.



(a) Operation mix by tier.

(b) Tier composition (1/2/3-step).

Figure 3.6: Dataset composition by difficulty tier (1/2/3 tool calls). Plots summarize per-tier operation mix and the stratified curriculum; deliverables are in Appendix A.

3.5 Conclusion

Acceptance criteria. Each example includes an executable tool-call sequence and a final-answer ground truth; therefore, we accept it only if deterministic replay reproduces the stored answer. Moreover, we bound difficulty by limiting traces to 1/2/3 tool calls and calibrate numeric sampling so values stay within declared ranges.

Reproducibility. As a result, we fix and report the tool vocabulary, generation configuration, and generation/split seeds, so downstream comparisons are attributable to learning dynamics rather than dataset noise. Finally, Chapter 4 distills this dataset into executable reasoning traces, and Chapter 6 optimizes verifier-checked correctness via GRPO.

Chapter 4

Distillation

4.1 Related Work

Positioning. Post-training for mathematical reasoning is commonly described as either (i) supervision on demonstrations from a stronger teacher (via distillation), or (ii) reinforcement learning with verifiable rewards (RLVR). In practice, pipelines combine both: distillation and SFT provide a stable "cold start" policy, and RLVR then improves verifier-checked correctness via exploration and reward assignment [12, 79, 84]. Therefore, this chapter treats distillation as data production: it converts the verified linear-algebra problems from Chapter 3 into multi-turn tool-use trajectories for SFT.

Main Purpose. Distillation is motivated by computational efficiency and capability transfer. DeepSeek-R1 reports that large-scale RL can elicit behaviors such as self-verification, reflection, and long reasoning traces, but smaller models struggle to rediscover these behaviors from scratch; consequently, distillation is used to transfer such patterns into smaller student models [12, 84]. Furthermore, Qwen3 argues that distillation can recover much of the benefit of multi-stage post-training at a fraction of the compute budget, which makes it a practical prerequisite when RLVR is unstable [79].

Format Objective. Because a single schema violation can block parsing, execution, and verification, we follow prior work that makes structure explicit. Firstly, ReAct frames tool use as an interleaving of reasoning and actions [81]; moreover, Toolformer shows this structure can be learned by inserting tool invocations directly into the training sequence [59]. Consequently, recent systems wrap generated text in explicit tags to keep generations parseable and verifiable at scale [12, 84]. Therefore, we specify a strict prompt contract (explicit tags and JSON tool calls) to maintain syntactic stability.

Rejection Sampling. Modern distillation increasingly relies on verification and filtering, retaining only traces that pass automatic checks [88]. Therefore, the dominant mechanism is rejection sampling [41] (as defined in Chapter 2). In addition, DeepSeek-Math [64] shows that RL can improve correctness under repeated sampling, and as a result we apply the same core idea in distillation by sampling multiple candidate trajectories per problem and retaining only those that are correct.

4.2 Overview

Motivation. In this thesis, distillation is used to compile executable supervision for tool-mediated linear algebra, rather than to match logits or hidden states (Definitions 2.1.11 and 2.1.2) [26, 20, 45]. Because downstream training consumes multi-turn tool-use traces, each example must include executed observations (e.g. matrix outputs) that the model can condition on in later turns. Therefore, a training example is treated as a replayable program over a fixed tool library.

Interface Contract. Therefore, we enforce a concrete interface contract for every retained trace: well-formed control tags, a schema-valid JSON `<tool_call>` with arguments, and tool outputs recorded as separate `tool`-role messages. Consequently, distillation acts as filtering under verification: we retain only traces that satisfy this contract and replay end-to-end, so the remaining non-correct cases reduce to verifier mismatches (or termination without a final answer).

Solution. We generate distillation data by running a teacher model inside a Read-Eval-Print Loop (REPL) tool environment, recording each tool call and tool output as a structured message trace. We sample multiple trajectories under stochastic decoding, producing a dataset of executable tool-use demonstrations. Figure 4.1 summarizes this process as a D.1 – D.4 loop (load → generate → execute/verify → publish).

Protocol. For reproducibility, we fix the input dataset split from Chapter 3, the teacher checkpoint (`Qwen/Qwen3-32B-FP8`) and decoding policy (including temperature, stop conditions, and a fixed turn/token budget), and the tool library. We then apply hard filter gates in order: the trace must be format-valid under the tag + JSON contract, executable under tool replay, and verifier-correct against the problem's ground truth. Finally, we enforce a semantic constraint: the final `<answer>` must be consistent with the last tool output, so the model cannot call a tool and then ignore its result.

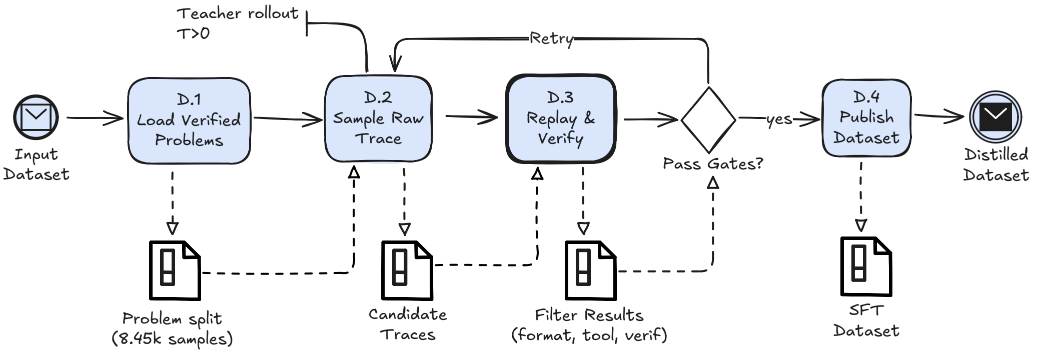


Figure 4.1: Distillation workflow schematic. Shows how verified synthetic problems are converted into multi-turn tool-use conversations and filtered by verifier checks.

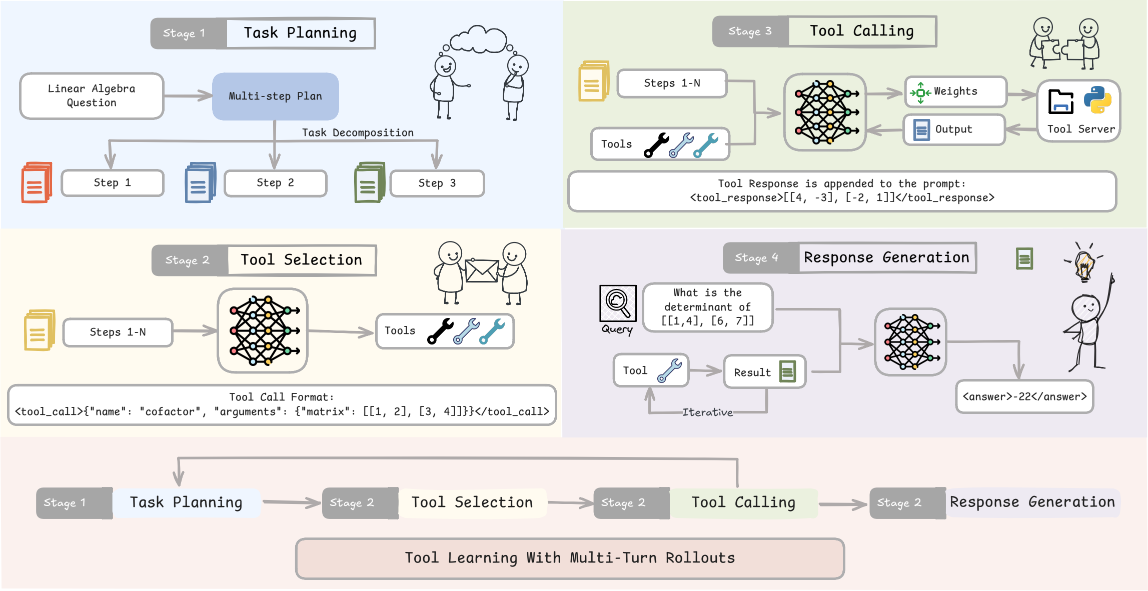
4.3 Model Architecture

Per-Turn Workflow. Figure 4.2 summarizes the per-turn tool-use cycle enforced by our distillation process. Standard distillation treats the teacher as a text generator; however, we run the teacher inside a REPL that executes the linear-algebra tools from Chapter 3. Consequently, each retained trajectory stores executable state that can be replayed during training and evaluation.

Chat Roles. We serialize each trajectory as an OpenAI-style message list with explicit roles: `system`, `user`, `assistant`, and `tool`. Tool outputs are stored as separate `tool`-role messages rather than being embedded in assistant text. The `assistant`-role messages correspond to the model-generated turns; moreover, during training and inference, the full message list is rendered into a single input string by applying the chat template (Definition 2.1.16).

Four-Phase Turn. Each tool-use turn follows a four-stage cycle that maps a natural-language query into actions and observations:

1. **Plan.** The model decomposes the query into a short sequence of subgoals.
2. **Select.** It chooses the appropriate tool for the next subgoal.
3. **Invoke.** It emits a <`tool_call`> with JSON arguments.
4. **Integrate.** It conditions on the tool output to produce the next <`tool_call`> or the final <`answer`>.



Tool Response is appended to the prompt:

```
<tool_response>[[4, -3], [-2, 1]]</tool_response>
```

Tool Call Format:

```
<tool_call>{"name": "cofactor", "arguments": {"matrix": [[1, 2], [3, 4]]}}</tool_call>
```

Figure 4.2: Four-phase tool-use turn: plan → select → invoke → integrate. The same strict tag and JSON schema contract is enforced in training and evaluation.

Delimited Turns. We enforce a strict per-turn contract via a fixed system prompt, reproduced in Appendix A, that specifies the tool set and required tags. Each assistant turn contains exactly one `<think>` block followed by either a single `<tool_call>` or a single `<answer>`. Moreover, we restrict trajectories to one tool call per turn so, tool replay remains simple and errors can be easily attributed (format violation vs. tool error vs. verifier mismatch).

Worked Example. Table 4.1 shows how these roles are applied in practice in a complete two-step example with two tool calls: the `assistant` emits `<tool_call>` actions, the `tool` role returns executed observations, and the final `<answer>` is taken directly from the last tool output. More examples are shown in Appendix B.

Table 4.1 also illustrates the behavioral constraints that make traces suitable for training. Firstly, the model avoids mental arithmetic by delegating each computation to a tool and, the `<think>` block is used only for planning and brief validation, while the tool interaction is expressed via a single `<tool_call>` per turn.

This structure is suitable for training in two ways: for SFT, the student can directly imitate the serialized `assistant` turns conditioned on the preceding `tool` observations. For GRPO, the system prompt enables the model to follow the format contract reliably.

Role	**Message content**
Example A (two-step; train[1268]).	
system	[System prompt excerpt: Use <think> then either <tool_call> or <answer>. Tool calls must be JSON with a declared tool name and arguments. Avoid manual computation; ground the final <answer> in the last tool output.]
user	Step 1: what is the transpose of A = [[2, 4, -2], [-3, 2, 4], [-4, 2, -3]]? Step 2: given matrix B = the result from step 1, find det(B).
assistant	<think>Okay, let's see. The user has a two-step problem. First, they need the transpose of matrix A, and then they need the determinant of that transposed matrix. So step 1 is to find the transpose of A. Matrix A is a 3x3 matrix. The transpose of a matrix is formed by swapping its rows and columns. The matrix_transpose tool should handle this. I'll call that tool with the given matrix A. Once I get the transposed matrix B, step 2 is to find its determinant. The determinant function requires a square matrix, which B should be since transposing a square matrix keeps it square. The determinant tool can compute that. So after getting B from step 1, I'll use the determinant tool on B to get the final answer.</think>

tool call	name=matrix_transpose
	arguments={"matrix": [[2, 4, -2], [-3, 2, 4], [-4, 2, -3]]}

tool	[[2, -3, -4], [4, 2, 2], [-2, 4, -3]]
assistant	<think>Okay, I need to find the determinant of the transposed matrix B, which is [[2, -3, -4], [4, 2, 2], [-2, 4, -3]]. The determinant tool can compute this. Since B is a 3x3 square matrix, the determinant function is applicable here. I'll call the determinant tool with matrix B as the argument.</think>

tool call	name=determinant
	arguments={"matrix": [[2, -3, -4], [4, 2, 2], [-2, 4, -3]]}

tool	-132.0
assistant	<think>Okay, so the user asked for two steps. First, transpose matrix A, then find the determinant of that transposed matrix. I used the matrix_transpose tool on A and got matrix B. Then, I applied the determinant tool on B and got -132.0. Since the determinant of the transpose is the same as the determinant of the original matrix, but the user wanted the determinant of B, which is the transpose, the answer is -132.0. No more steps needed, so I'll present that as the final answer.</think> <answer>-132.0</answer>

Table 4.1: A two-step tool-use trace illustrating the thought–action–observation structure, sequential tool chaining through intermediate observations, and a brief self-verification check.

4.4 Optimizations/Analysis

Figure 4.3 presents the distillation loop at a high level, illustrating the system components (teacher, tool execution environment, and verifier) and the verification filter that decides whether a trace is kept or discarded. In addition, Algorithm 4.1 illustrates the fine-grained step-by-step process for trajectory generation.

Rejection Sampling. Figure 4.3 summarizes distillation as a verification filter. Firstly, each problem provides hidden symbolic ground truth that is never shown to the teacher. The teacher is sampled stochastically ($T > 0$) to propose a trace, and the environment executes each `<tool_call>` to produce observations. Consequently, we keep a trajectory only if it satisfies the format contract, executes without tool errors, and matches the ground truth; otherwise we discard it and resample.

Distillation Algorithm. Two approaches were evaluated for handling malformed teacher outputs: self-correction via diagnostic messages, and rejection sampling that discards invalid trajectories and resamples.

In practice, diagnostic messages polluted the conversation, because the model sometimes referenced them in later turns. For example, when a turn omitted a required tag, we appended a diagnostic user message such as "Format error: emit exactly one `<tool_call>` or `<answer>` after `<think>`" and prompted the teacher to retry; however, the subsequent assistant messages could sometimes reference these diagnostic artifacts. Therefore, we use rejection sampling rather than in-context repair.

Algorithm 4.1 provides the fine-grained control flow used to mine trajectories under the verification filter from Figure 4.3.

Probabilistic Trajectory Mining. We rely on probabilistic trajectory mining: by raising the sampling temperature to $T = 0.6$ and $top\_p = 0.95$, we treat the teacher model

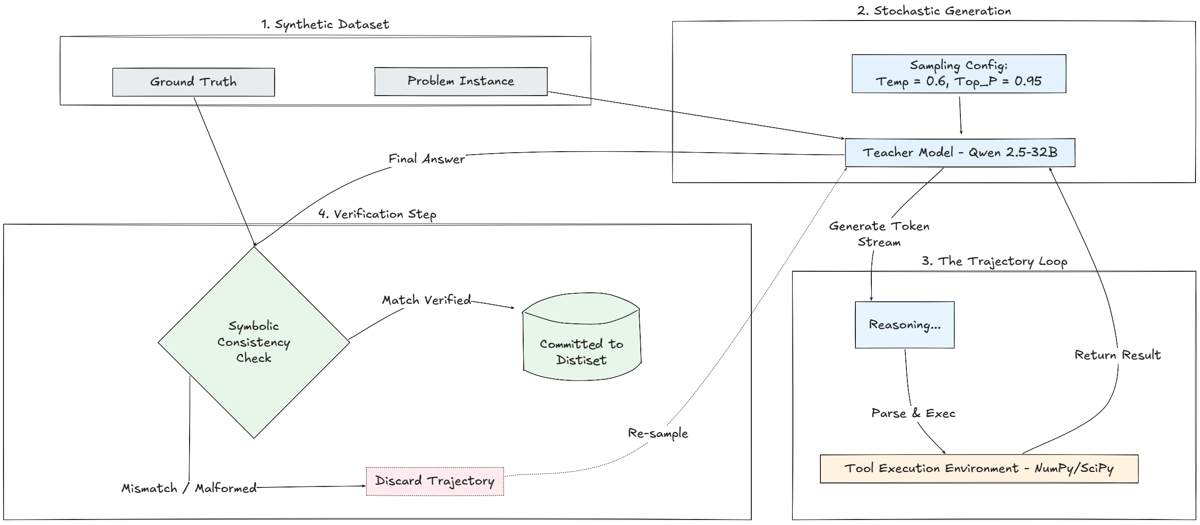


Figure 4.3: Rejection-sampling distillation loop. Only verifier-correct, schema-valid teacher trajectories are kept as supervision.

```
procedure DistillWithToolUse($\mathcal{D}_{\text{problems}}$, $\pi_\theta$, $\mathcal{T}$, $T_{\max}$, $B$)
    $\mathcal{D}_{\text{distill}} \leftarrow [\,]$                                    ▷ accepted trajectories
    while HasNextBatch($\mathcal{D}_{\text{problems}}$) do
        $\mathcal{B} \leftarrow$ NextBatch($\mathcal{D}_{\text{problems}}, B$)
        for all $query \in \mathcal{B}$ in parallel do
            $ctx \leftarrow$ InitializeState($query$)          ▷ system prompt + tool schema + user query
            for $t \leftarrow 1$ to $T_{\max}$ do
                $resp \leftarrow \pi_\theta(ctx)$                  ▷ model output: tool call or final answer
                if IsToolCall($resp$) then
                    $(obs, ok) \leftarrow$ ExecuteTool($\mathcal{T}, resp$)        ▷ observation + success flag
                    if $ok$ then
                        $ctx \leftarrow$ UpdateState($ctx, resp, obs$)
                    end if
                else
                    $ctx \leftarrow$ UpdateState($ctx, resp$)                  ▷ record final answer
                    break
                end if
            end for
            $\tau \leftarrow$ ExtractTrajectory($ctx$)                        ▷ interaction trajectory
            $is\_correct \leftarrow Validate(query, \tau)$                  ▷ binary correctness label
            Append($\mathcal{D}_{\text{distill}}, (\tau, is\_correct)$)
        end for
    end while
    return $\mathcal{D}_{\text{distill}}$
end procedure
```

Algorithm 4.1. Distillation with multi-turn tool use: sample problems, generate multi-turn tool-use trajectories (up to $T_{\max}$), validate, and retain accepted traces.

as a stochastic generator under temperature sampling (Chapter 2/Section 2.1.3). Therefore, we roll out trajectories until the teacher naturally converges on a schema-valid path. Because validity constraints compound across turns, this rejection-sampling strategy replaces in-context repair with repeated sampling, retaining only trajectories that are already format-valid for downstream SFT.

Parallel Mining. To make large-scale trajectory mining practical, we batch problems and generate assistant turns in parallel using a vLLM-backed inference server. Moreover, we use the `distilabel` library [90] to orchestrate concurrent rollouts, improving throughput via Ray-based parallel scheduling across available GPUs [46].

4.5 Experiments

Experimental Overview. This section validates the distilled tool-use traces for downstream SFT by analyzing context-window constraints, length-based filtering, and post-hoc curation effects on the final SFT-ready dataset. All experiments were executed on Runpod [56], an on-demand GPU platform for renting GPU instances; we detail the instance configuration and software environment later in this section.

4.5.1 Dataset Cleaning and Filtering

Failure Breakdown. Table 4.2 summarizes dataset retention across two stages: (i) failures during trajectory mining and verification, and (ii) post-hoc curation that removes remaining data-quality issues.

The mining run stores 8,376 traces, of which 8,325 are verifier-correct; the remaining 51 traces are non-correct (termination without a final answer or a verifier mismatch) and are excluded from the SFT subset. Because format validity and tool replay are enforced as validation gates during distillation, format and tool-runtime failures are not present in the generated dataset.

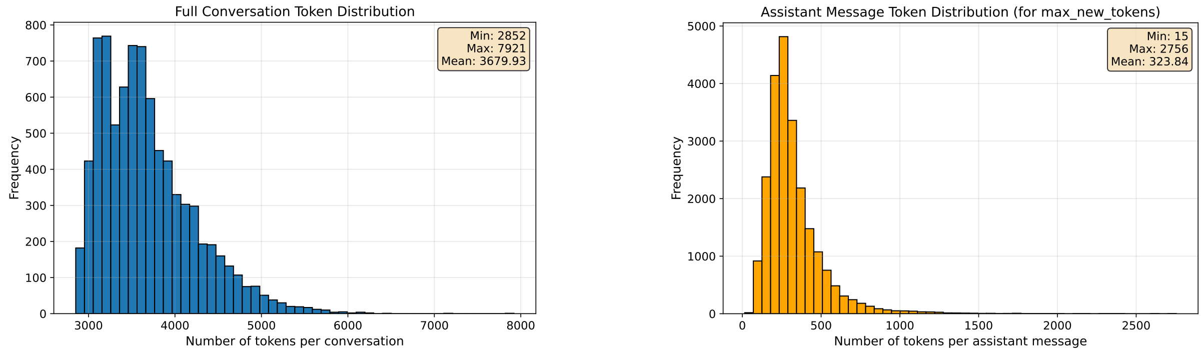
The post-hoc filters then remove 677 traces (union of issues) for interface inconsistencies and long-context outliers, yielding a cleaned dataset of 7,648 examples. We enforce an upper bound of 800 tokens per turn; empirically, above this threshold the model frequently begins to perform mental arithmetic, which is undesirable because it can introduce avoidable errors.

Context Window Pressure. Figure 4.4 shows token-length histograms at both the per-turn (assistant) and full-conversation levels. During manual inspection, we observed that the model frequently performs mental arithmetic in a subset of traces, and this behavior is strongly correlated with unusually long assistant turns.

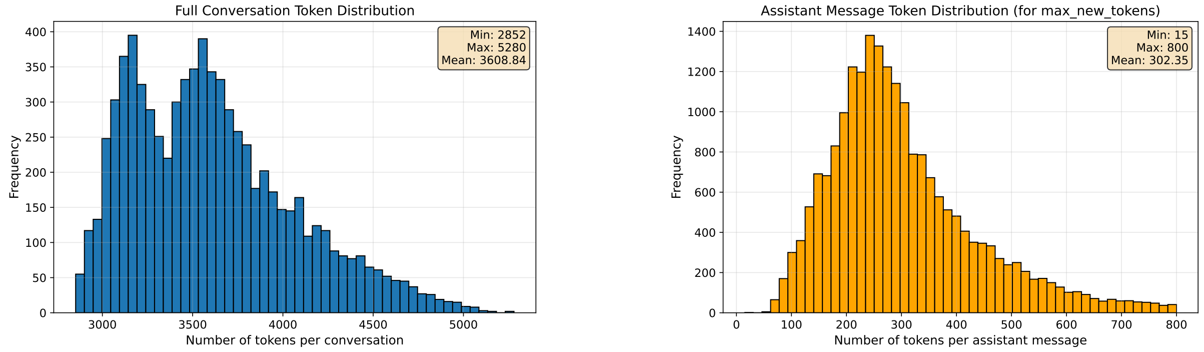
Therefore, we remove any conversation in which at least one assistant turn exceeds 800 tokens (Qwen2.5-3B tokenizer), removing 536 conversations. This filtering improves the downstream SFT signal by discouraging undesirable reasoning patterns; moreover, it removes extreme length outliers (e.g. > 7000 tokens), reducing VRAM pressure and enabling local SFT runs rather than cloud-only training.

Outcome category	Count	Rate (%)
Mining outcomes (stored distiset, n=8376)		
Format violation (malformed turn)	0	0.00
Tool runtime error	0	0.00
No final answer (did not terminate)	17	0.20
Verifier mismatch (answered, but incorrect)	34	0.41
Post-hoc validation filters (verifier-correct subset, n=8325)		
Answer–tool mismatch	63	0.76
Repeated tool calls	33	0.40
Tool-sequence mismatch	64	0.77
Message-count mismatch	36	0.43
Token threshold exceeded	536	6.44
Duplicate `<think>` blocks	65	0.78
Stage retention		
Retained after mining (excluded: 51; out of n=8376)	8325	99.39
Retained after post-hoc (excluded: 677; out of n=8325)	7648	91.87
Total excluded (mining+post-hoc)	728	8.69
Final retained (global; out of n=8376)	7648	91.31

Table 4.2: Distillation quality breakdown: mining outcomes from the stored distiset, post-hoc validation filters, and final retention rates.



(a) Conversation length (raw). **(b)** Assistant length (raw).



(c) Conversation length (filtered). **(d)** Assistant length (filtered).

Figure 4.4: Trace length distributions (raw vs. filtered). Filtering removes long-context outliers to fit within the training context window.

4.5.2 Infrastructure and Scalability

Limitations. Local development on consumer hardware (NVIDIA RTX 4090, 24GB VRAM) revealed significant limitations. The GPU could not load the teacher model (`Qwen2.5- 32B`) at a useful quantization level without offloading layers to the CPU; because of this, inference slowed to minutes per generation and became impractical to use. Consequently, we resorted to a cloud infrastructure to run the experiments.

Cloud Architecture. Figure 4.5 illustrates the core design of the RunPod [56] infrastructure. We ran the distillation pipeline on a single NVIDIA H100 (80GB PCIe) GPU, which allowed us to load `Qwen/Qwen3-32B-FP8` entirely into GPU memory. By using FP8, we retained sufficient numerical precision while staying within the memory budget. Moreover, `vLLM` [36] enabled high-throughput inference by leveraging Flash Attention 2 [11]. In contrast to local inference (minutes per trajectory), the cloud setup processed one batch of 24 requests in approximately 2 minutes and 30 seconds. The pipeline starts the inference server as a separate process, and it supports both vLLM [36] and llama.cpp [42].

Key Hyperparameters. The main mining run uses `Qwen/Qwen3-32B-FP8` [54] as the teacher with stochastic decoding at `temperature=0.6` and `top_p=0.95`. We cap each

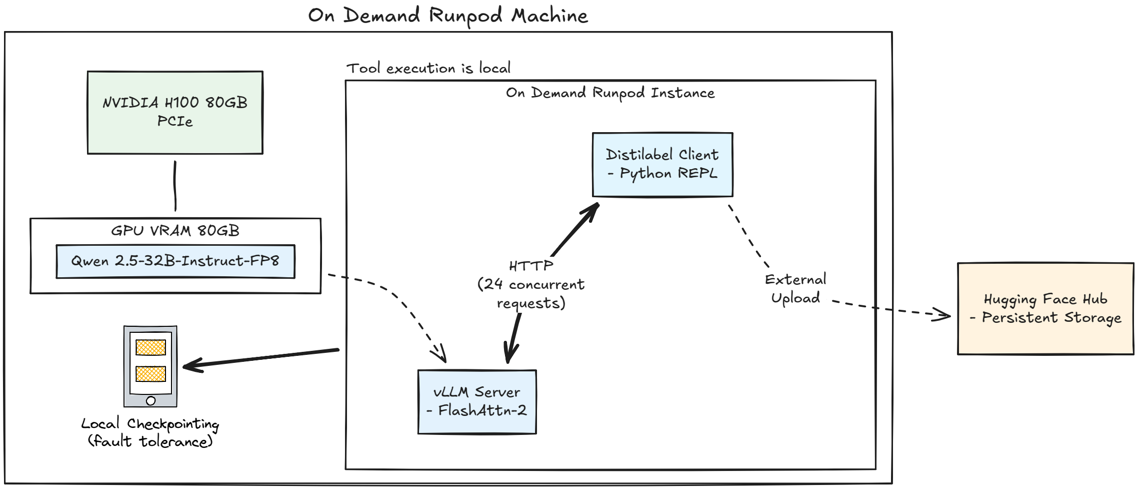

On Demand Runpod Machine
NVIDIA H100 80GB PCIe
GPU VRAM 80GB
Qwen 2.5-32B-Instruct-FP8
Local Checkpointing (fault tolerance)
Tool execution is local
On Demand Runpod Instance
Distilabel Client - Python REPL
HTTP (24 concurrent requests)
vLLM Server - FlashAttn-2
External Upload
Hugging Face Hub - Persistent Storage

Figure 4.5: RunPod compute infrastructure for distillation. For failure tolerance, intermediate artifacts are checkpointed to local disk, enabling the run to resume after interruptions. The run completed in 14 hours and produced 8,376 verified trajectories.

trajectory at `n_turns=5` turns and `max_new_tokens=4096` tokens per turn, while enforcing a trajectory-wide context limit of `max_model_len=24576`. Moreover, we use explicit stop tokens (i.e. `</answer>`, `</tool_call>`) to terminate generations cleanly once the final answer is produced, ensuring reliable output boundaries for parsing and execution. To maximize throughput, we mine in parallel with `input_batch_size=24` requests via `vLLM`.

Operational Costs. The experiment ran for approximately 14 hours, costing ≈$25. We used the `distilabel` framework for fault tolerance: in the event of failures, intermediate artifacts were persisted to local disk, allowing the run to resume from the last completed step. Furthermore, the REPL and inference engine ran on the same host, so tool execution did not require any network communication.

4.6 Conclusion

This chapter connects the problem curriculum from Chapter 3 to the training data used in Chapter 5. By pairing a Python REPL with a strict tool-use schema, we turn the teacher from a text generator into an agent whose intermediate steps are grounded in executed tool outputs. Moreover, by preferring rejection sampling over in-context self-correction, we avoid training on failed or context-polluting conversations and instead mine trajectories that are valid under repeated sampling.

In the final analysis, each retained example is a structured conversation (`messages`) with tool definitions and a ground-truth correctness label. Lastly, the conversations are executable under the tool library and verifier-correct, yielding a clean supervision dataset for fine-tuning.

Chapter 5

SFT

5.1 Related Work

Positioning. SFT converts a general-purpose model into a policy that can reliably operate under a fixed format contract [86]. DeepSeek-R1 [12] and Qwen3 [79] therefore recommend a cold-start SFT phase before RL, because it increases the rate of valid tool calls and, consequently, improves format adherence – a hard constraint for effective tool use. Therefore, distillation-driven instruction tuning provides a mechanism for transferring teacher tool-calling behavior into smaller models via SFT [68, 85].

Format Bottleneck. Tool calling is a multi-turn protocol, so format failures compound across steps rather than affecting only a single-turn answer. Definition 2.1.24 formalizes this: a per-turn validity rate p implies at most p^N probability of an executable N-turn trajectory [15]. Consequently, malformed rollouts yield sparse rewards under exploration [12, 79]. Therefore, cold-start SFT is needed to teach the tool-call template before RL, because it raises p and keeps a non-trivial fraction of rollouts executable.

Solution. Tokenization defines the discrete symbols a model can emit. However, standard subword tokenizers such as BPE [62] and SentencePiece [35] can fragment structural delimiters into multi-token sequences. Because of this, tool calling becomes brittle: a single token-level error can invalidate an entire turn and prevent tool execution. Recent work therefore argues for tokenizer–model adaptation in domains where structured control tags define the interface format [16, 33, 65]. We address this bottleneck by introducing atomic special tokens for interface tags.

Format Drift. We merge LoRA weights into the base model to obtain a standalone checkpoint for downstream GRPO serving. This merge is also operationally required: vLLM assumes a single base model with fixed tensor shapes, and adapters that modify `embed_tokens` or `lm_head` (e.g. via vocabulary expansion) break this assumption [71]. One critical problem is that merging after vocabulary resize can disturb the output distribution in the resized `lm_head`, reducing the emission probability of newly added control tokens and degrading format adherence [21, 62]. Therefore, this motivates the use of a post-merge repair step to re-calibrate the I/O layers before running GRPO [9].

5.2 Overview

Motivation. This chapter uses SFT to align the model with the tool-use format contract (Appendix A), producing a checkpoint that reliably emits tool-call traces from the distilled dataset in Chapter 4. Because the base model only predicts next tokens and does not follow instructions, this stage aims for interface adherence. Consequently, given that format compliance is already high (approx. 96% – see Table 6.2), Chapter 6 treats this checkpoint as a strong baseline for RL and tests for incremental reasoning improvements without re-learning syntax.

Solution. Figure 5.1 outlines the end-to-end workflow. Firstly, `S.1` adds special tokens and resizes the vocabulary so the format tags are learnable. Secondly, `S.2` performs LoRA-based SFT on the distilled traces to learn multi-turn tool use. By capping each trajectory to 8,000 tokens, training becomes feasible locally, on a 24GB memory GPU. Thirdly, `S.3` merges the adapters to produce a standalone checkpoint, which is a requirement for the RL phase. Fourthly, `S.4` repairs the input/output layers so the resized token embeddings remain optimized after merging. Finally, `S.5` evaluates the resulting checkpoint on a held-out test set.

Evaluation. We evaluate with greedy decoding on a fixed test dataset and treat end-to-end correctness (strict parsing, tool execution, and ground-truth verification) as the primary metric, using training loss only as a diagnostic to guide optimization progress. We run evaluation in parallel with vLLM for throughput; minor nondeterminism can still appear even under greedy decoding due to parallel execution, so we report the mean of 5 evaluation runs. Therefore, we select the checkpoint that maximizes format validity and report the paired correctness result in Section 5.5.3.

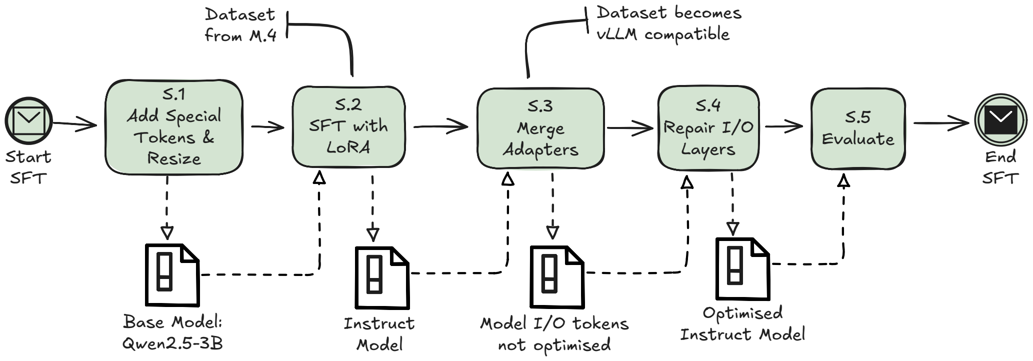


Figure 5.1: SFT workflow schematic. Illustrates how supervised fine-tuning aligns the model to the tool-use format contract to be used for downstream RL.

5.3 Model Architecture

The core motivation for this pipeline is to satisfy the model constraints required by the RL phase. We outline below the rationale around each decision.

Format Contract. Each assistant turn must output exactly one planning block (i.e. `<think>...</think>`), followed by exactly one action: either a single `<tool_call>` containing one JSON object, or a single final `<answer>`. Any tokens outside these boundaries represent a contract violation.

Control Tags. We implement the control tags as special tokens rather than plain text, so `<think>`, `<tool_call>`, and `<answer>` are emitted as single tokens instead of a sequence of fragments such as <, `tool`, `_call`, >. As a consequence, using single-token tags reduces syntax fragility under greedy and sampled decoding, which is especially important for the Qwen2.5-3B model as it uses a byte-level BPE tokenizer [55]. One additional benefit to using tags is to enforce delineation so the model learns to separate internal reasoning from tool interactions and the final answer, which simplifies parsing and supports robust RL.

In addition, stop strings ensure decoding halts immediately once the model emits a closing tag (typically `</ tool_call>` or `</answer>`). This enables reliable parsing; otherwise the model may overrun the boundary and emit extra tokens that break the contract.

SFT with LoRA. We begin from the Qwen2.5-3B base model and perform SFT with LoRA adapters attached to a subset of the linear layers. This enables single-GPU training, since full-parameter fine-tuning would otherwise be impractical on one GPU. The SFT with LoRA setup uses Unsloth [10] and TRL [72] for single-GPU end-to-end training.

Merged Adapters. GRPO rollouts are served with vLLM, which can only load a checkpoint with fixed tensor shapes (as discussed in [71]). After we expand the vocabulary, `embed_tokens` and `lm_head` change shape, and vLLM does not support adapters that override these modules at runtime (through `modules_to_save`). Therefore, we merge the LoRA deltas into the base model and use a standard merged checkpoint for RL.

Repair I/O Layers. Merging after vocabulary resize shifts the `lm_head` logits, reducing the probability of the newly added control tokens. Moreover, since `modules_to_save` is used, the trained I/O layers are not merged and must be re-trained. Therefore, we run a short repair epoch that freezes the transformer body and trains only `embed_tokens` and `lm_head` to re-align the interface.

5.4 Analysis

Chat Template. As noted in Chapter 2, the chat template fixes how the message list is serialized into a single token string for training and decoding. We therefore treat it as a fixed part of the model specification and use the same template, without modification, across all training phases, evaluation, and deployment.

System Prompt Standardization. The system prompt is the reference definition of the interface contract, documented in Appendix A. As a result, we standardize it during dataset preparation, ensuring every conversation begins with the same tool-use rules and tag constraints. The only difference across phases is that distillation uses chain-of-thought examples to teach explicit format compliance, while SFT and GRPO do not strictly require them because the standardized prompt already fixes the interface and the objective loss function reinforces adherence.

Qualitative Analysis. The primary focus of this phase was to teach instruction-following and contract adherence. Before SFT, the `Qwen-2.5-3B` base model mainly treats the prompt as a textual prefix and generates textbook-like problems or incoherent LaTeX fragments[1]. Consequently, after fine-tuning, the model adheres to the format contract, emits parseable tool calls and final answers, and solves queries through step-by-step tool invocations.

Plain-Text Inefficiency. Learning the interface as plain text is inefficient because the loss is spread across the full conversation instead of focusing on the core format tags that enable tool use. Because of this, a 3B model is more likely to miss the small set of schema-valid trajectories under sampling during RL, making rewards very sparse. The ablation study in Section 5.5.2 supports this claim, showing that plain-text tags fail to produce valid tool calls and confirming that special tokens are necessary for effective tool use.

Trade-off. The frozen-output-layer checkpoint is a deliberate compromise: it locks the resized input/output layers so the tool-use tags remain stable under vLLM, but it also caps the accuracy ceiling because the head cannot keep improving beyond the SFT checkpoint. Because of this, the model gets a reliable format contract for RL while sacrificing some output flexibility. In consequence, the base model learns to reliably emit executable tool calls under sampling, so reward signals remain usable for RL even though reasoning improvements must pass through a fixed head.

---

[1] For examples of malformed base model outputs see this link.

5.5 Experiments

5.5.1 Experimental Setup

Dataset Format and Configuration. We transform the distilled traces from Chapter 4 into a tool-calling SFT format by following [14], and we train on the resulting dataset which includes 7,648 trajectories. The dataset explicitly requires the model to emit the new special tokens (`<think>`, `<tool_call>`, `<answer>`) introduced by the vocabulary expansion. For reference, we summarize the training hyperparameters in Table 5.1.

Evaluation Protocol. We evaluate SFT end-to-end by following the same approach from Figure 4.2: generate a turn, parse it, execute a tool call if present, and continue until a final answer is produced. During evaluation we use greedy decoding to ensure that the generated data reflects interface stability rather than sampling noise. Therefore, a trajectory only counts as successful if it stays parseable across turns and terminates in a verifier-checkable `<answer>`.

Metrics. We decompose performance into interface and outcome metrics, following Section 2.4. Therefore, we report (i) format validity via format accuracy, answer-tag accuracy, and tool-call-tag accuracy, and (ii) outcome accuracy via answer accuracy. This separation makes it easier to identify failures: the model must learn the format interface before it can improve answer correctness.

Checkpoint Selection. For model selection, we choose the checkpoint with highest format validity on the validation set and report the paired correctness under the same (greedy) decoding protocol. We also report correctness alongside format.

Hyperparameter	**Value**
Base Model	Qwen/Qwen2.5-3B
Precision	bfloat16 (Flash Attention 2)
LoRA Rank (r)	32
LoRA Alpha (α)	64
LoRA Dropout	0.1
Target Modules	All Linear Layers (q, k, v, o, gate, up, down)
Batch Size	16 (Gradient Accumulation)
Learning Rate	2×10^{-5}
LR Scheduler	Cosine with Minimum LR (2×10^{-6})
Warmup Ratio	0.03
Optimizer	AdamW (Fused)
Epochs	1
Max Sequence Length	8192

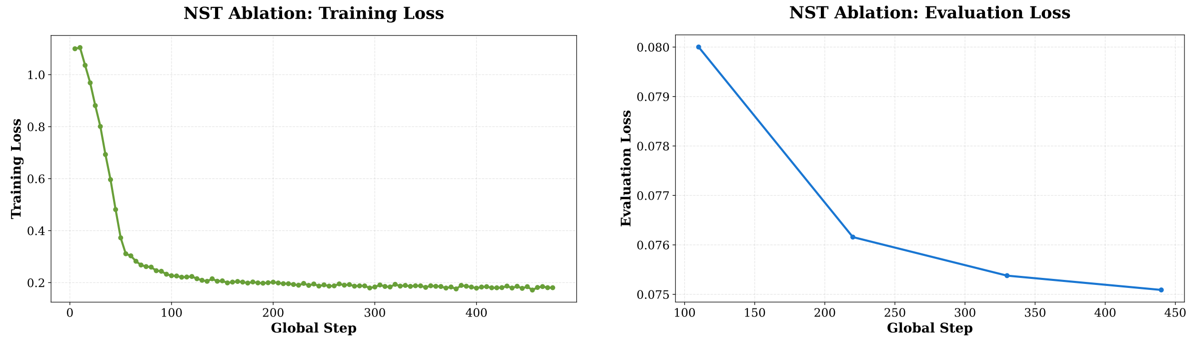
Table 5.1: Hyperparameters used during the SFT phase.

5.5.2 Ablation: No Special Tokens for Interface Tags

Plain-Text Tags. In this experiment, we tested whether the tool-use interface can be learned without extending the tokenizer vocabulary, by performing a run where `<think>`, `<tool_call>`, and `<answer>` remain ordinary plain-text strings rather than atomic special tokens. In this context, the model must reproduce these delimiters as multi-token sequences, which makes exact tag emission a more challenging objective for a 3B model.

Evidence. Table 5.2 reports the interface and outcome metrics for the no-special-token run, with the full trace available in W&B (`sft-ablation-nst`). Figure 5.2 shows that both training and evaluation loss decrease over the training steps. However, tool-call tag accuracy remains 0% across the entire epoch, so tool execution never occurs and end-to-end correctness does not improve. This ablation therefore indicates that special tokens are necessary for effective tool use because they stabilize the format contract.

Possible Failure Cause. When interface tags are plain text, the model must emit multi-token delimiters exactly, and a single token error breaks parsing. The tokenizer also fragments tags, so several sub-tokens are required to produce a valid `<tool_call>` block. As a result, loss can decrease while format validity remains near zero, because near-miss strings are rewarded even though they never trigger tool execution.



(a) Training loss

(b) Evaluation loss

Figure 5.2: NST ablation loss curves during SFT: training loss (left) and evaluation loss (right).

Evaluation	Answer	Format	Answer Tag	Tool-call Tag
Step 110	2.31%	1.39%	3.85%	0.00%
Step 220	3.08%	3.06%	8.46%	0.00%
Step 330	3.46%	2.92%	8.08%	0.00%
Step 440	3.46%	3.06%	8.46%	0.00%

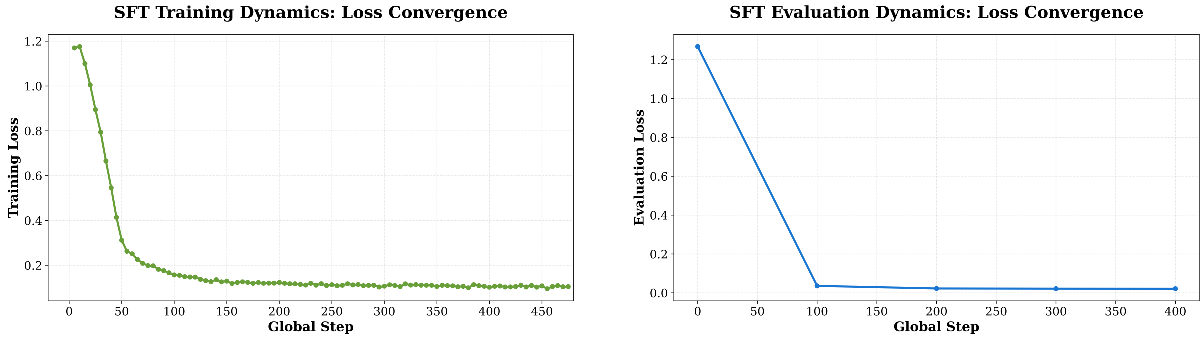
Table 5.2: Ablation results when the interface tags are not added as special tokens and must be produced as plain text. All evaluation steps use 260 samples.

5.5.3 SFT Loss and Accuracy Trends

SFT Results. We report final evaluation results for the produced model: Figures 5.3a and 5.3b show loss convergence, while Table 5.3 summarizes the merged checkpoint. Learning is not uniform. Format adherence (Figure 5.3c) rises sharply in the first 100 steps, enabling the pre-trained model to rapidly learn to use the tool-use protocol; once tags are stable, correctness (Figure 5.3d) rises in parallel and the validation curve improves simultaneously. Sample SFT vs RL tool-use traces are shown in Appendix B.

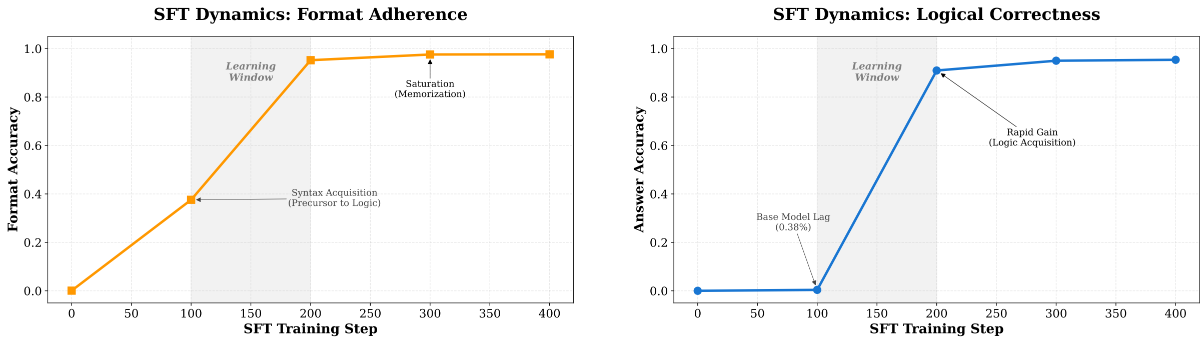
Note that the learning curves for the LoRA and merged checkpoints are very similar. For brevity, we report only the LoRA metrics; merged checkpoint results are linked in Appendix A together with corresponding W&B runs. In practice, one epoch suffices: by step 400 the best checkpoint reaches near-optimal performance with 95% answer accuracy and 97% format accuracy. On the held-out test set, end-to-end correctness is 89.87%. The final test-set results appear in Table 6.2.

Compounding risk. As already noted, multi-turn tool use compounds per-turn format failures [15]: even a 95% per-turn format accuracy drops to about 81% over four turns ($0.95^4 \approx 0.81$). Under GRPO sampling, this bound can fall further. Nonetheless, these results indicate that format accuracy remains high enough to work with the fixed-head architecture, resulting in an interface that is stable enough to support effective GRPO training.



(a) Training Loss (Global Convergence)

(b) Evaluation Loss (Generalization)



(c) Format Adherence

(d) Logical Correctness

Figure 5.3: SFT training dynamics across loss, format adherence, and correctness. All evaluation steps use 520 samples.

Evaluation	Answer	Format	Answer Tag	Tool-call Tag
Step 0	0.00%	0.07%	0.00%	0.11%
Step 100	0.38%	37.57%	0.38%	58.59%
Step 200	90.96%	95.21%	90.96%	97.61%
Step 300	95.00%	97.57%	95.00%	99.02%
Step 400	95.38%	97.64%	95.77%	98.70%

Table 5.3: Merged checkpoint evaluation metrics across format validity and correctness. All evaluation steps use 520 samples.

5.6 Conclusion

Result. The key result is that the format contract can be compiled into a generic text model: training on the distilled multi-turn traces from Chapter 4 converts the `Qwen-2.5-3B` base model into a policy that emits parseable tool calls and a final answer. Therefore, the injected special tokens act as stable tool-use delimiters, and the merge-repair workflow yields a standalone checkpoint that preserves those tokens while re-calibrating the resized I/O layers. In the final analysis, this stabilizes the output interface so downstream training does not need to rediscover basic syntax.

Bridge. With a reliable format-compliant baseline, the next phase asks whether GRPO can add measurable reasoning gains beyond an already strong SFT checkpoint. Because SFT reaches near-ceiling accuracy, headroom is limited, so we frame GRPO as a robustness and stress test, especially since exploration noise can degrade format stability. In conclusion, Chapter 6 applies GRPO under the same end-to-end tool-format protocol to assess deviations from the SFT baseline.

Chapter 6

Group Sequence Policy Optimization

6.1 Related Work

Reinforcement Learning. GRPO was introduced in DeepSeekMath [64] and later reused in DeepSeek-V3 [13] and DeepSeek-R1 [12]. Unlike actor – critic methods such as PPO [61], GRPO estimates advantages by comparing multiple rollouts from the same prompt, removing the need to train a separate value model. In PPO, the value model serves as a learned baseline for advantage estimation. Consequently, recent research [84] shows that GRPO is often used in constrained settings because it reduces memory usage while retaining PPO-style stabilization mechanisms such as clipping.

GRPO Instability. A common critique of token-level importance weighting in vanilla GRPO is that verifier rewards are often sparse, which can lead to instability. When a single terminal reward is assigned across many tokens, the resulting gradients can have high variance. Consequently, sequence-level variants such as GSPO [89] compute importance ratios from the likelihood of the full sampled sequence and apply clipping to improve reward granularity and stabilize long-horizon rollouts. Therefore, we use GSPO because our rewards are trajectory-level over many turns.

Verifier-Based Rewards. RLVR replaces subjective preference judgments with deterministic reward signals derived from a verifier, which is especially suitable in domains such as mathematics and code where correctness can be checked automatically [64, 12]. The verifier replays tool calls and checks the final answer, tying rewards to execution outcomes rather than a learned preference model. This provides an objective signal for scaling multi-turn agents. Nonetheless, verifier-based rewards remain vulnerable to reward hacking, where the policy optimizes the signal without the intended behaviour.

Tau-Bench. Tau-bench [80] frames tool use as a multi-turn interaction loop where success is judged by a deterministic end state rather than a free-form explanation. It also proposes pass^k to measure stability over repeated trials, which motivates evaluating tool agents under multiple rollouts rather than relying on a single greedy trajectory. Consequently, we mirror Tau-bench's setup by using an end-state verifier to score multi-turn tool trajectories, implemented on top of the Art library [24].

6.2 Overview

Motivation. Chapter 5 shows that distillation and SFT can compile the tool-format contract into a standalone checkpoint that reliably solves linear algebra problems. Because the reward is only defined for executable trajectories, GSPO is only meaningful once this format-compliant baseline exists. Therefore, this stage asks whether GSPO can add measurable verifier-checked reasoning gains beyond a near-ceiling SFT policy while stress-testing format stability under stochastic exploration.

Challenges. Most challenges were engineering rather than conceptual. Because tool-calling RL frameworks are still maturing and end-to-end GSPO-with-LoRA support is limited, framework selection was difficult[1]. In addition, stable long-horizon training required tuning sensitive hyperparameters, so we needed a library that implements GSPO.

Solution. Nonetheless, we adopted the Art library [24], which satisfied these requirements. It supports multi-turn rollout training with LoRA and implements GSPO as a GRPO variant that applies sequence-level stabilization [89]. Therefore, we run RL in a closed linear-algebra tool environment and summarize the loop in Figure 6.1 as four control points: tuning stability hyperparameters, defining the reward function, scheduling a difficulty curriculum, and selecting the best checkpoint over 3 epochs.

Protocol. GSPO rollouts are sampled stochastically to create within-group diversity (necessary for effective learning), while evaluation uses greedy decoding on a fixed held-out set. Because concurrent execution introduces run stochasticity, we report each evaluation point as the mean over three repeated runs. We treat held-out greedy end-to-end correctness as the primary performance metric and track format validity, tool success, and reward allocation as auxiliary diagnostics.

---

[1] For example, our initial attempt to use verl [66] was blocked because its current implementation did not support LoRA adapters.

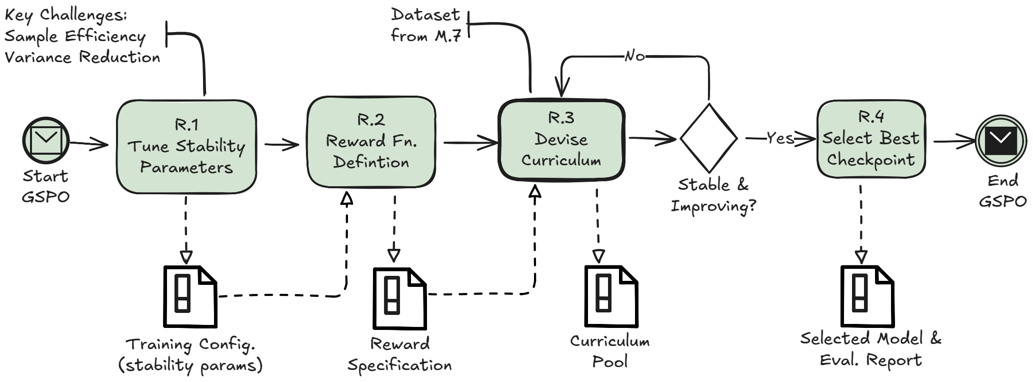


Figure 6.1: Highlights GSPO training: define rewards, tune stability, run curriculum, and select the best checkpoint under a deterministic parse → execute → verify loop.

6.3 Architecture

Architecture Goal. This section explains how we generate and score multi-turn rollouts, and then how we convert grouped trajectories into GSPO updates under stochastic sampling. We start from the 3B checkpoint compiled in Chapter 5; because the tool interface must remain vLLM-usable, we keep the resized input/output layers (`embed_tokens` and `lm_head`) frozen and route learning through LoRA adapters. Therefore, we treat the SFT checkpoint as the stability baseline and test whether GSPO can improve evaluation accuracy by stress-testing the frozen-head architecture.

Rollout Generation. Figure 6.2 illustrates rollout generation. On the left, a linear-algebra environment, a tool-calling agent, and an LLM execute multi-turn episodes and log traces, tool interactions, and verifier rewards as grouped rollouts. To keep rollout throughput high, we parallelize environment execution in Python and run decoding through vLLM [36] with `FlashAttention` [11]. On the right, the loop collects trajectory groups and handles evaluation and checkpointing.

The output of rollout generation is a set of trajectory groups, where each group

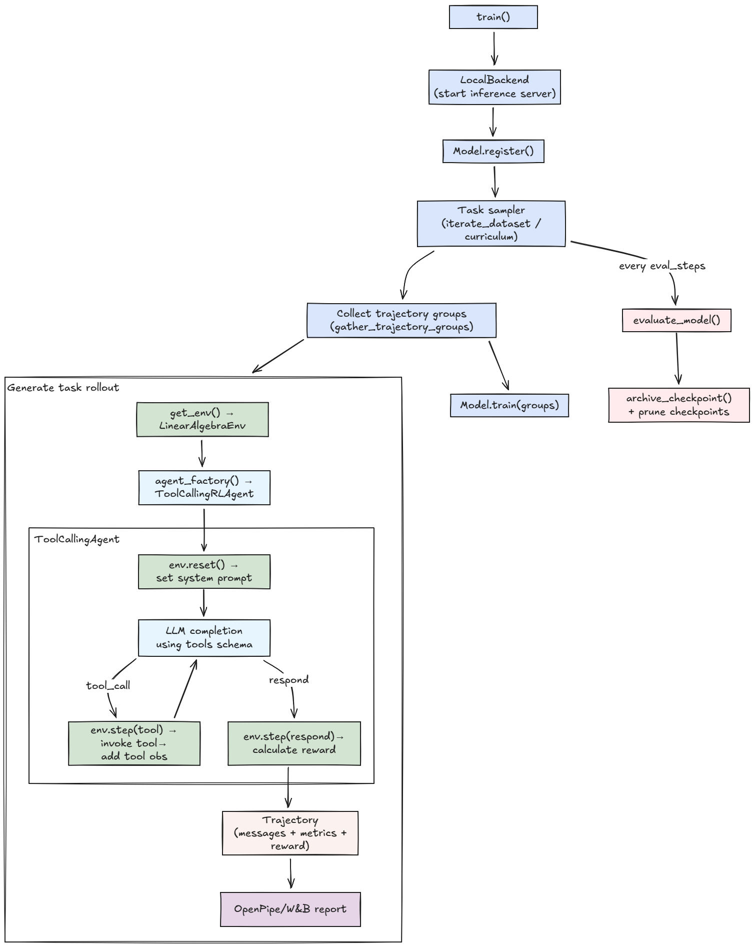

train()
LocalBackend (start inference server)
Model.register()
Task sampler (iterate_dataset / curriculum)
every eval_steps
Collect trajectory groups (gather_trajectory_groups)
evaluate_model()
Generate task rollout
Model.train(groups)
archive_checkpoint() + prune checkpoints
get_env() → LinearAlgebraEnv
agent_factory() → ToolCallingRLAgent
ToolCallingAgent
env.reset() → set system prompt
LLM completion using tools schema
tool_call
respond
env.step(tool) → invoke tool→ add tool obs
env.step(respond)→ calculate reward
Trajectory (messages + metrics + reward)
OpenPipe/W&B report

Figure 6.2: Rollout generation loop: an environment, a tool-calling agent, and an LLM interact to produce multi-turn rollouts, which are then scored to compute advantages for GSPO updates.

contains G sampled responses to the same unique prompt; the group becomes the basic unit for computing advantages in GSPO, which determines the learning signal.

Training Pipeline. Figure 6.3 illustrates an overview of GSPO training using the trajectory groups gathered during the rollout phase. Firstly, the application calls `model.train`, which tokenizes trajectories, computes group-relative advantages, and packs tensors for training. Then, the training service enters training mode, pauses vLLM to free GPU memory, and runs multiple optimization steps. Finally, it resumes vLLM, reloads the updated weights, and reports the training metrics.

Compute Rationale. Reinforcement learning fine-tuning is often memory-bound. One optimization is to omit the KL-divergence term, since recent work shows GSPO remains effective without this regularizer [28]; consequently, we do not keep a separate reference policy in memory, which further reduces memory usage. On top of this, training uses LoRA adapters, so only the adapter parameters are trainable; consequently, optimizer state and gradient buffers are stored only for the adapters, which substantially reduces GPU memory and makes single-GPU training feasible.

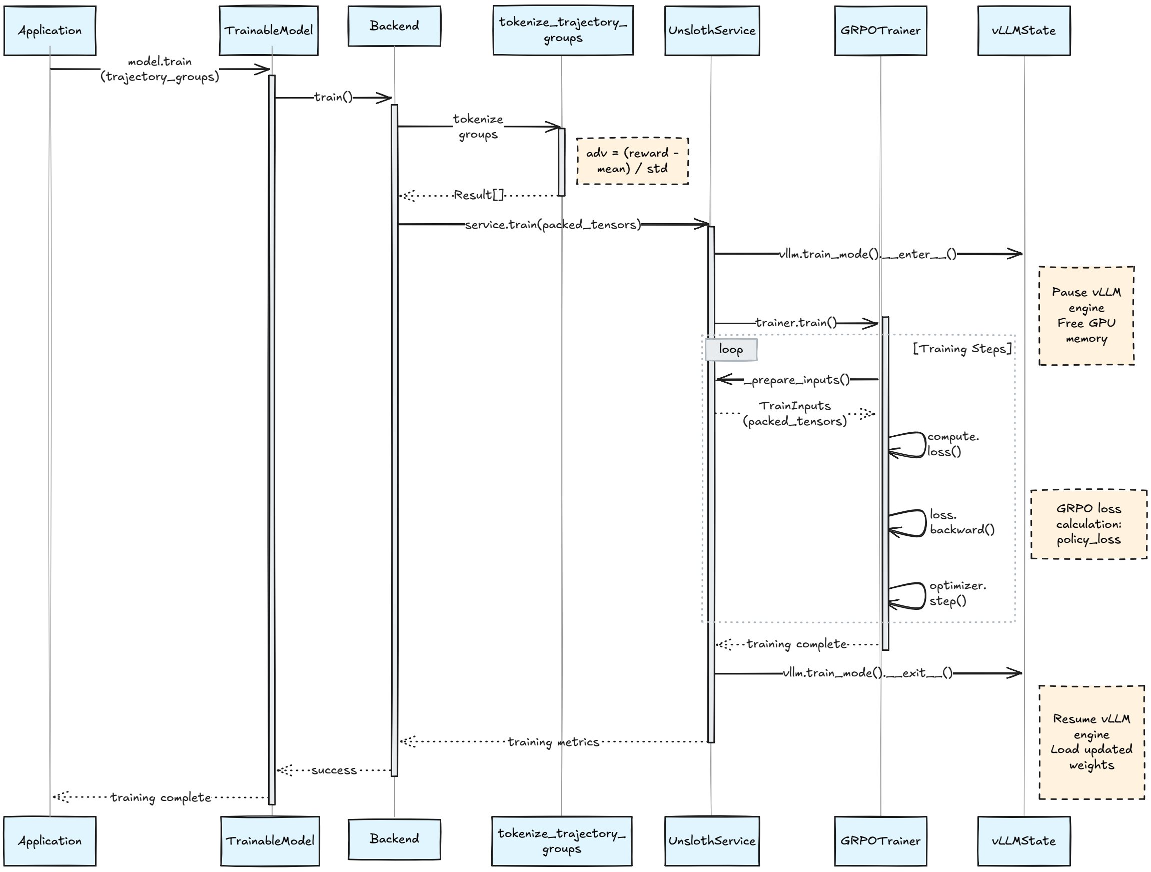

Application
TrainableModel
Backend
tokenize_trajectory_groups
UnslothService
GRPOTrainer
vLLMState
model.train (trajectory_groups)
train()
tokenize groups
adv = (reward - mean) / std
Result[]
service.train(packed_tensors)
vllm.train_mode().__enter__()
Pause vLLM engine Free GPU memory
trainer.train()
loop
[Training Steps]
_prepare_inputs()
TrainInputs (packed_tensors)
compute. loss()
loss. backward()
GRPO loss calculation: policy_loss
optimizer. step()
training complete
vllm.train_mode().__exit__()
Resume vLLM engine Load updated weights
training metrics
success
training complete
Application
TrainableModel
Backend
tokenize_trajectory_groups
UnslothService
GRPOTrainer
vLLMState

Figure 6.3: GSPO training loop. Samples grouped rollouts, computes group-relative advantages, and applies clipped policy updates. We purposely omit the kl divergence term [28].

6.4 Optimizations/Analysis

6.4.1 Training Stability

Algorithm. Algorithm 6.1 shows the GSPO loop that links rollout sampling to policy updates, including our choice to omit the KL-divergence regularizer. For each batch of prompts, we sample a group of G trajectories per prompt from the rollout policy $\pi_{\theta_{\text{old}}}$, score each trajectory with the verifier, and compute group-relative advantages by normalizing rewards within the group. We then update the policy using a clipped, sequence-level importance ratio, which improves stability when rewards are assigned to entire multi-turn trajectories. Here, $\pi_{\theta_{\text{old}}}(\cdot \mid x)$ denotes the full response distribution conditioned on prompt x, so sampling yields $\{y_i\}_{i=1}^{G}$.

Stability. RL fine-tuning can become unstable, leading to exploding gradients and rapid policy drift that degrades both correctness and format compliance. We stabilized training by tuning the learning rate and clipping hyperparameters (ϵ), and by monitoring training diagnostics such as gradient norms and reward variance. We also shaped rewards to bias sampling toward valid, executable trajectories, with components: correctness (1.0), format validity (0.1), tool success (0.1), and an inefficiency penalty (-0.1), normalized by 1.2. The reward function definition is available in Appendix A.

```
Input initial policy model $\pi_{\theta_{\text{init}}}$; task prompts $\mathcal{D}$; hyperparameters $\epsilon$, $\mu$; group size $G$
policy model $\pi_\theta \leftarrow \pi_{\theta_{\text{init}}}$
for iteration = 1, . . . , I do
    for step = 1, . . . , M do
        Sample a batch $\mathcal{D}_b$ from $\mathcal{D}$
        Rollout policy $\pi_{\theta_{\text{old}}} \leftarrow \pi_\theta$

        # Rollout generation (Fig. 6.2).
        For each prompt $x \in \mathcal{D}_b$, sample a group $\{y_i\}_{i=1}^{G} \sim \pi_{\theta_{\text{old}}}(\cdot \mid x)$

        # Training pipeline (Fig. 6.3).
        Compute rewards $\{r_i\}_{i=1}^{G}$ and advantages $\hat{A}_i \leftarrow \frac{r_i - \text{mean}(\{r_j\}_{j=1}^{G})}{\text{std}(\{r_j\}_{j=1}^{G})}$
        for GSPO epoch = 1, . . . , $\mu$ do
            Compute $s_i(\theta) \leftarrow \left(\frac{\pi_\theta(y_i|x)}{\pi_{\theta_{\text{old}}}(y_i|x)}\right)^{1/|y_i|}$
            Compute $L \leftarrow \frac{1}{G}\sum_{i=1}^{G} \min(s_i(\theta)\hat{A}_i,\ \text{clip}(s_i(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_i)$
            Update $\pi_\theta$ by maximizing $L$
        end for
    end for
end for
Output $\pi_\theta$
```

Algorithm 6.1. Optimized GSPO. We omit the KL term from Chapter 2, following evidence this improves results [28]. Combines rollout generation (Figure 6.2) and training loop (Figure 6.3).

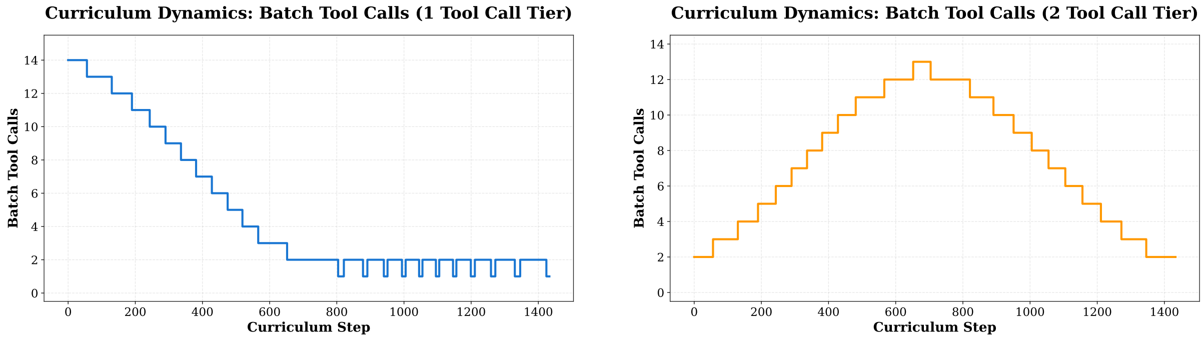
6.5 Experiments

Setup. We train GSPO for three epochs with learning rate 10^{-6}, sequence-level clipping $\epsilon \in [3 \times 10^{-4}, 4 \times 10^{-4}]$, and no KL-to-reference regularization ($\beta = 0$). For each prompt, we sample $G = 8$ trajectories and process 16 groups per update step, yielding an effective batch size of 128 samples. Trajectories are decoded with $T = 1.0$. We update only LoRA adapters ($r = 32$, $\alpha = 64$) on top of the Chapter 5 checkpoint.

6.5.1 Curriculum

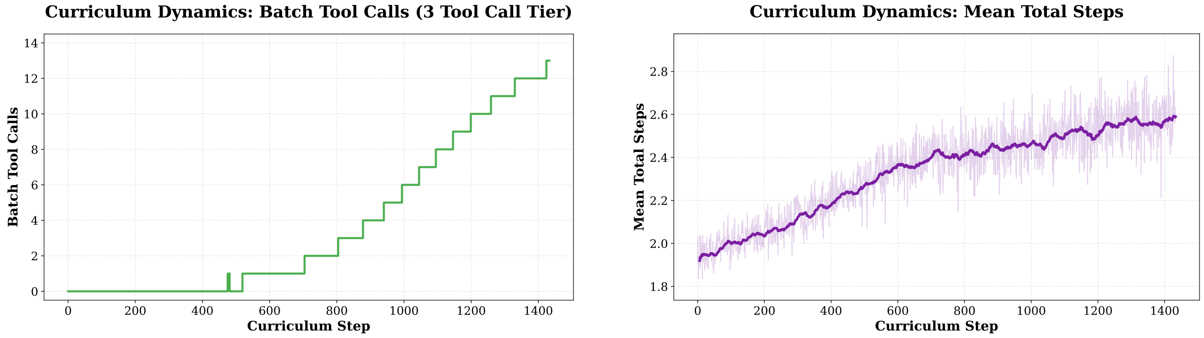
Curriculum. We adjust training difficulty over time by changing the sampling distribution over tool-call tiers, where a tier is the number of tool invocations in a problem sample (between 1 and 3 tool calls). Each additional tool call adds another multi-turn step that must retain format validity. This curriculum increases difficulty over time, which stabilizes training by preventing the reward signal from being too sparse early on.

Early in training, the sampler draws mostly 1-tool-call problems; at fixed intervals, probability mass shifts to the 2-tool-call tier and later to the 3-tool-call tier. This staged schedule gives the model time to adapt to longer-horizon traces, avoiding the case where problems are too difficult early on. Figure 6.4 reports tier usage over time and shows that mean steps increase as the curriculum shifts toward higher tiers.



(a) 1 tool call tier

(b) 2 tool call tier



(c) 3 tool call tier

(d) Mean total steps

Figure 6.4: Curriculum scheduling during GRPO. Tracks the training-time distribution of sampled tool-call tiers (1–3 tool calls) and the resulting mean trajectory length (total steps).

6.5.2 Results

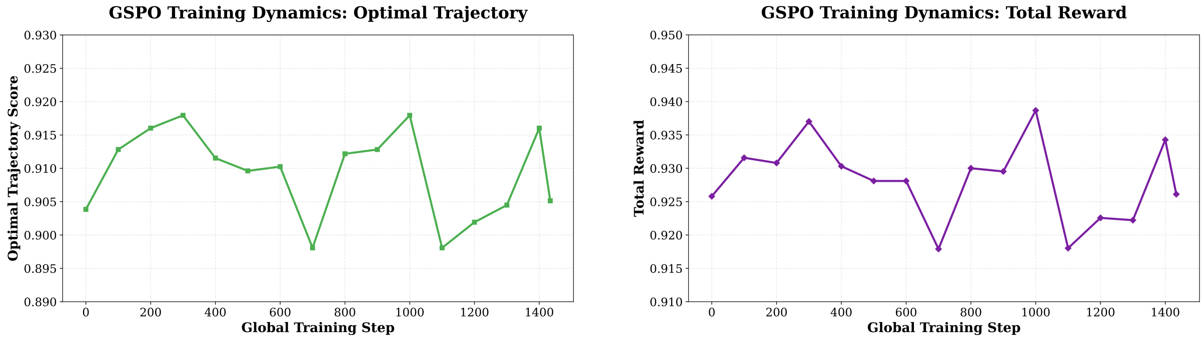
Protocol. Following the evaluation methodology in Section 2.4, evaluation fixes a 520-sample held-out split and uses greedy decoding, while GSPO rollouts are sampled stochastically during training. Because concurrent execution introduces noise due to non-determinism, we report each evaluation point as the mean over three independent runs. We refer to Section 2.4 for the definitions of all evaluation metrics.

Evaluation Results. Figure 6.5 reports GSPO evaluation metrics. For checkpoint selection, we choose the model with maximum `optimal_trajectory` (Step 1000), and then report correctness and format validity; total reward is optimized during training but not used for selection because it includes auxiliary components beyond end-to-end correctness (such as the turn deviation penalty). At Step 1000, correctness improves but format validity remains near the SFT ceiling, which limits `optimal_trajectory`, because the frozen input/output layers likely limit direct format gains[2].

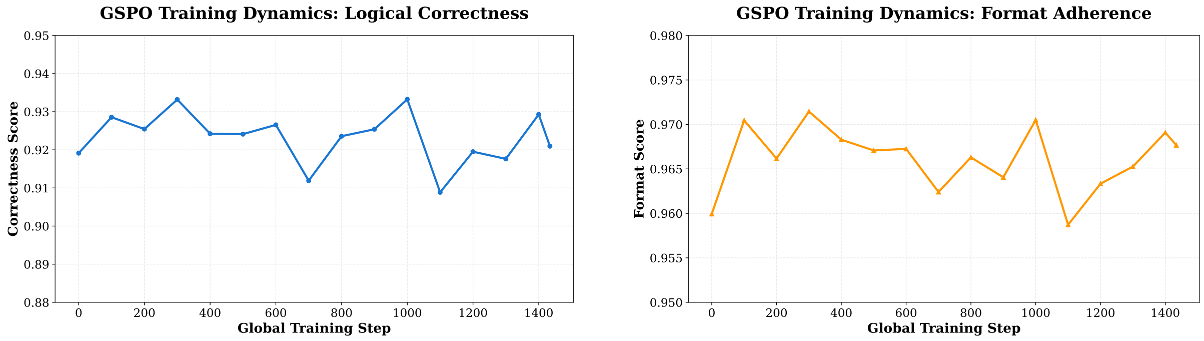
At the selected Step 1000 checkpoint (peak `optimal_trajectory`), verifier-checked correctness improves by about 1.4% absolute on the 520-sample evaluation set (roughly 7 additional solved problems). Format validity holds steady at the SFT ceiling, so end-to-end success remains capped. Appendix B compares SFT vs GRPO traces.

---

[2] In our setup, these layers are frozen due to serving/training constraints discussed in Chapter 5.



(a) Optimal Trajectory (Aggregated Performance) **(b)** Total Reward



(c) Logical Correctness (Reasoning) **(d)** Format Adherence (Syntax)

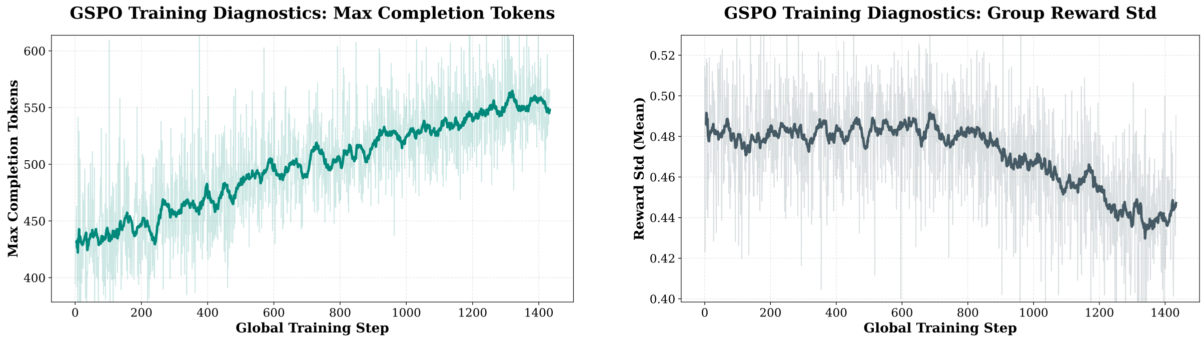
Figure 6.5: GSPO training dynamics under a frozen head. Shows held-out greedy metrics over a 3-epoch training run.

Operational Costs. We run GSPO on a single RTX 6000 Ada GPU (48GB VRAM) for approximately 2 days and 9 hours (57 hours). At an on-demand rate of $0.77/hour, the GPU time totals approximately $44; in practice, we used $50 overall.

Stability Diagnostics. Figure 6.6a shows that rollouts get longer during training, which is consistent with the model producing longer traces before giving the final answer. In addition, Figure 6.6b illustrates advantage diversity at each step: the advantage distribution must remain diverse to avoid spiky updates that can destabilize previously learned behavior. The non-zero group reward standard deviation indicates that rollouts vary in quality, providing a usable learning signal.

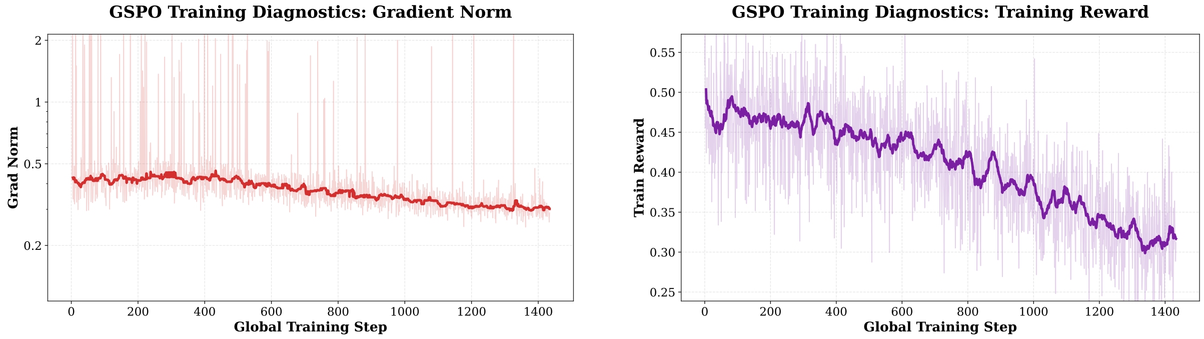
Furthermore, Figure 6.6c shows occasional spikes, indicating higher-variance updates. Nonetheless, after gradient clipping, training remains stable: although the maximum unclipped gradient norm peaks at 12.53, the validation metrics in Figure 6.5 do not degrade.

The training reward in Figure 6.6d trends downward and remains noisy in part because the curriculum shifts the training distribution: later batches are systematically harder and tend to require longer trajectories (Figure 6.6a). Consequently, step and inefficiency penalties can pull the mean `train/reward` down even if held-out correctness improves.



(a) Max completion tokens. As training progresses, generation length increases.

(b) Group reward std. Indicates sufficient group advantage diversity for healthy training.



(c) Gradient norm (y-axis clipped to the 2nd–98th percentile; max unclipped value = 12.53).

(d) Training reward. Reward in [0, 1], used to calculate advantages.

Figure 6.6: GSPO training-time curves are reported as diagnostic signals for stability. Full W&B traces available in Appendix A.

Performance Ceiling. The dominant limitation in this phase is the frozen output projection (`lm_head`) under stochastic rollouts. Because vLLM does not support LoRA training with learnable input/output embeddings via `modules_to_save`, these layers remain fixed; consequently, the policy cannot increase the logit gap that makes required control tokens (e.g., <`tool_call`>) reliably beat nearby alternatives.

Therefore, format adherence acts as a ceiling on end-to-end tool use (Figure 6.5d) even if adapters may learn better decisions as response length increases. Under this constraint, the observed ~96% format-validity rate appears to be a practical limit for the 3B model in this setup; consequently, it upper-bounds `optimal_trajectory`. Nonetheless, larger models or an unfrozen head may raise overall performance even further.

Error Breakdown. Table 6.1 summarizes the model's failure categories on the test set and complements the accuracy metrics in Table 6.2. Because of minor non-determinism from parallelism and vLLM, we report the mean (N = 1,560; 3 runs × 520 samples). To handle samples with multiple failure reasons, we apply a priority order, as listed top-to-bottom in Table 6.1, to assign a failure category to each trace. Consequently, each failed sample is counted once and totals remain comparable across the two tables.

Errors are dominated by answer-tag failures and format-bad cases in both models, while forced stops and invalid trajectories are rare. In particular, GSPO reduces error rates for missing or unparseable <`answer`> tags. On the other hand, format-bad and turn-deviation rise slightly; therefore, the bottleneck remains format control (tag structure and tool-call count), not tool execution. In the final analysis, there are no exclusive tool-fail or incorrect samples. The evaluation traces and the script used to generate Table 6.1 are available under "Failure Breakdown" in Appendix A.1. The final test-set metrics for SFT and GSPO are summarized in Table 6.2.

	SFT		GSPO		Example
Error category	Count	Rate (%)	Count	Rate (%)	Failure Reason
Exclusive failure reasons (priority order, top = higher priority)					
Forced stop	3	0.19	3	0.19	Reached maximum allowed tool calls
Valid trajectory = 0	16	1.03	16	1.03	Trace invalid after parsing (e.g. truncated tags)
Answer tag missing	60	3.85	43	2.76	Final missing <`answer`> tag
Answer unparseable	20	1.28	14	0.90	Final <`answer`> not parseable as a value
Format bad	47	3.01	58	3.72	Malformed <`tool_call`>/<`answer`> tags
Tool fail	0	0.00	0	0.00	Tool execution fails
Incorrect	0	0.00	0	0.00	Answer provided but wrong
Turn deviation	12	0.77	18	1.15	Tool-call count: expected N, got $N \pm 1$
Total errors	158	10.13	152	9.74	Sum of exclusive categories

Table 6.1: Error breakdown (SFT vs GSPO). Failure breakdown for SFT vs GSPO on the same 520-sample split, aggregated across three runs (N = 1,560). A single failure reason is assigned when `optimal_trajectory = 0` using the fixed priority order; rates use error counts divided by 1,560, so the total error rate equals 1 − `optimal_trajectory`. Examples for each failure mode are available in Appendix C.

Metric	**LinAlgZero-SFT**	**LinAlgZero-GRPO**
Optimal Trajectory	89.87%	90.26%
Correctness	91.86%	92.63%
Format Validity	96.15%	96.66%
Tool Success	100.00%	100.00%

Table 6.2: Evaluation metrics for the test set (Definitions 2.4.2–2.4.5 in Chapter 2).

6.6 Conclusion

Outcome. This chapter shows that GSPO improves `optimal_trajectory` over an already strong SFT baseline. We select the Step 1000 checkpoint by maximal held-out `optimal_trajectory`; on the test set, the selected model reaches `optimal_trajectory` = 90.26% (+0.39%) and verifier-checked correctness = 92.63% (+0.77%). Nonetheless, format validity remains near the SFT ceiling, which caps overall gains, because the frozen output head limits direct improvements in format adherence.

Ceilings And Next Steps. The remaining headroom is primarily architectural: with a frozen input/output interface[3], GSPO has limited ability to improve tool-call syntax under sampling. Consequently, format validity saturates near ~96% in this setup, even as adapters learn better decisions and rollouts become longer. The next step is to scale to a larger model or use an unfrozen-head configuration so policy improvements could potentially translate into higher test-set accuracy.

---

[3] In our setup, we freeze these layers because vLLM does not support LoRA training with `modules_to_save`.

Chapter 7

Conclusion

7.1 Ethical Considerations

Loss of learning. LLM assistance in mathematics can undermine learning when it replaces, rather than supports, the student's own problem-solving. On the one hand, free-form explanations can expose intermediate decisions and therefore provide some assurance about how an answer was reached. On the other hand, tool-using agents often reduce transparency: because key operations are delegated to external tools and the model mainly schedules calls, the fine-grained reasoning behind each choice can be hard to infer. Consequently, heavy reliance on automatically generated solutions can discourage active engagement and harm long-term learning.

To mitigate this risk, prior work [37] recommends using such systems only with practices that support active learning, including additional verification, algorithmic literacy, and human-in-the-loop workflows. Concretely, verification can use deterministic libraries (e.g. SymPy) or formal proof assistants to check the work, while human-in-the-loop workflows keep a user engaged by guiding the resolution process. Crucially, algorithmic literacy provides the mathematical competence needed to verify the trajectory rather than accepting it at face value. Therefore, users are encouraged to treat the system as an assistant rather than an authority, which helps preserve learning through active engagement.

Compute Footprint. Beyond pretraining, post-training stages such as SFT and reinforcement learning can add significant compute overhead. However, satisfying these requirements is not always feasible outside well-resourced labs. Because of this, experimentation can be constrained by hardware budgets and time constraints, which can in turn limit iteration speed and broader participation.

We delegate exact linear algebra computations to a Python runtime, so the model learns tool use rather than long free-form reasoning. To make this feasible under tight compute and memory constraints, we use parameter-efficient fine-tuning (LoRA) for both SFT and RL, and lower-precision distillation (e.g., FP8), which together reduce training memory and overall cost. These optimizations allow the pipeline to fit within single-machine GPU budgets. Nonetheless, further efficiency improvements suggested by prior work include power capping [44], model compression [57], and quantization [32], which can often reduce cost with limited quality loss.

Solution Hacking. A recurring risk when granting LLMs agency is solution hacking: the model reaches a rewardable output by exploiting the system rather than following the intended workflow [47]. For example, during tool use, an agent may ignore tool results and instead iterate against verifier feedback to "guess" an answer. Consequently, outputs can look correct even when the underlying process is wrong.

Deterministic symbolic solvers (e.g. SymPy) make arithmetic correctness a characteristic of the tool, which localizes failures to planning errors or tool misuse. However, this does not eliminate the risk of optimizing against the reward signal in unintended ways. One partial mitigation is to add process-level checks or rewards (e.g., validating intermediate tool inputs/outputs and enforcing invariants), which can make reward hacking harder by tying reward to verifiable intermediate steps; nonetheless, these signals must be designed carefully because they can introduce new failure modes. Consequently, users still need enough algorithmic literacy to verify the trajectory, especially when using such systems for important decisions.

7.2 Future Work & Limitations

Synthetic Dataset Generation. Future work could extend the curriculum with problems that require verified intermediate steps and deeper composition. For example, it could add SVD-derived tasks and least-squares problems, where correctness depends on intermediate values and precondition checks (e.g. full-rank inputs). Furthermore, the generator could introduce branching dependencies so that later checks depend on multiple earlier values, which diversifies the tool-use trajectories.

Human preferences alignment. This extension could collect pairwise preferences over distilled transcripts by sampling multiple trajectories for the same prompt and labeling a chosen vs rejected completion, then training with DPO [38]. Concretely, these preferences can target tool-use attributes such as interaction style (e.g. when to call tools and how many calls per turn), helpfulness (clear explanations that support learning), and correctness – not only the final answer.

Process-level rewards. This extension adds verifiable intermediate states that enable dense, stepwise rewards during tool-use trajectories. Firstly, the reward can target concrete sub-steps such as valid tool-call syntax and explicit precondition checks (e.g. input dimensions, rank conditions) before an operation is executed. Consequently, these dense signals give finer control over the behavior being encouraged, rather than relying only on the final answer outcome.

7.3 Conclusions

Thesis Question. This thesis asked whether a compact `Qwen2.5-3B` base model can solve linear algebra problems by producing executable tool-use trajectories rather than free-form responses. We show that this is feasible and yields strong post-SFT performance on end-to-end tool-use metrics.

Contributions. The central contribution is methodological: we build and validate an end-to-end pipeline for linear algebra tool-use. Taken together, these contributions answer the four research questions in Section 1.1. The main contributions are:

- A reproducible pipeline that unifies generation, distillation, SFT, and GSPO under a single tool-use protocol (Chapters 3–6).
- An entropy-controlled synthetic linear-algebra generator that produces arbitrarily many problems with controlled compositional depth (Chapter 3).
- A distillation pipeline that converts verified problems into multi-turn, executable tool-use traces, yielding trajectories that serve as SFT supervision (Chapter 4).
- An SFT baseline for `Qwen2.5-3B` that stabilizes the special-token interface (format validity and tool-call execution) in a standalone checkpoint (Chapter 5).
- A GSPO stress test on a near-ceiling SFT checkpoint showing stable RL fine-tuning under the tool-use protocol (Chapter 6).

Results. In the final analysis, the best test metrics are: optimal-trajectory accuracy 90.26%, verifier-checked correctness 92.63%, and format accuracy 96.66% (Table 6.2).

Lessons Learned. The main practical lessons from this project are:

- Familiarity with the end-to-end workflow required to teach a base model to plan and reason using tools, which gives insight into reproducing established post-training workflows.
- Experience running distillation and RL experiments in a cloud environment, including reproducibility, cost, and fault tolerance, which teaches iteration under compute constraints.
- Logging and inspection of generated traces using tools such as `OpenPipe` [48] and `distilabel` [90] to identify data-quality issues, which is a prerequisite for stable, effective training.
- Writing and maintaining a large technical document that reports the end-to-end workflow, which supports reproducibility, reflection, and long-term learning.

Closing. Our main difference from prior work is that we study tool-use learning in a closed linear-algebra domain with a strict parse – execute – verify loop. For engineers, the implication is practical: we provide a reproducible recipe for training a small, self-hostable tool-using model with good performance, rather than relying on closed-source large-model APIs. For students, the takeaway is pedagocial: it shows how a pretrained base model can be adapted into a full tool-calling agent through dataset design, distillation, SFT, and RL, rather than only prompt engineering.

Moreover, our results suggest that strong end-to-end tool-use performance is achievable with modest resources; in our setup, distillation and GSPO fit within a ~ $75 cloud budget. Nonetheless, extending these results to broader mathematical domains, longer tool chains, and more complex problem dependency dynamics remains future work.

Bibliography

[1] 2026. Dirichlet distribution. Page Version ID: 1331574579.

[2] Alon Albalak, Duy Phung, Nathan Lile, Rafael Rafailov, Kanishk Gandhi, Louis Castricato, Anikait Singh, Chase Blagden, Violet Xiang, Dakota Mahan, and Nick Haber. 2025. Big-Math: A Large-Scale, High-Quality Math Dataset for Reinforcement Learning in Language Models. GSCC: 0000010 2025-06-25T15:17:29.640Z 0.56 arXiv:2502.17387 [cs].

[3] Art of Problem Solving. Art of Problem Solving. Public problem repository and community (accessed 2025-12-23).

[4] arXiv. arxiv. Open-access preprint repository (accessed 2025-12-23).

[5] Samy Bengio, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. Scheduled sampling for sequence prediction with recurrent neural networks.

[6] Christopher M. Bishop. 2006. *Pattern Recognition and Machine Learning*. Springer, New York, NY.

[7] Bernd Bohnet, Rumen Dangovski, Kevin Swersky, Sherry Moore, Arslan Chaudhry, Kathleen Kenealy, and Noah Fiedel. 2025. A Comparative Analysis of LLM Adaptation: SFT, LoRA, and ICL in Data-Scarce Scenarios. ArXiv:2511.00130 [cs].

[8] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

[9] Gautier Dagan, Gabriel Synnaeve, and Baptiste Rozière. 2024. Getting the most out of your tokenizer for pre-training and domain adaptation. ArXiv:2402.01035 [cs].

[10] Michael Han Daniel Han and Unsloth team. 2023. Unsloth.

[11] Tri Dao. 2023. Flashattention-2: Faster attention with better parallelism and work partitioning.

[12] DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang,

Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, et al. 2025. DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning. ArXiv:2501.12948 [cs].

[13] DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, et al. 2025. DeepSeek-V3 Technical Report. ArXiv:2412.19437 [cs].

[14] Huggingface docs. Dataset formats and types.

[15] Nouha Dziri, Ximing Lu, Melanie Sclar, Xiang Lorraine Li, Liwei Jiang, Bill Yuchen Lin, Peter West, Chandra Bhagavatula, Ronan Le Bras, Jena D. Hwang, Soumya Sanyal, Sean Welleck, Xiang Ren, Allyson Ettinger, Zaid Harchaoui, and Yejin Choi. 2023. Faith and Fate: Limits of Transformers on Compositionality. ArXiv:2305.18654 [cs].

[16] Mete Erdogan, Abhiram Gorle, Shubham Chandak, Mert Pilanci, and Tsachy Weissman. 2026. An Information-Theoretic Perspective on LLM Tokenizers. ArXiv:2601.09039 [cs] version: 1.

[17] Tommaso Furlanello, Zachary C. Lipton, Michael Tschannen, Laurent Itti, and Anima Anandkumar. 2018. Born again neural networks.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[19] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023. Pal: Program-aided language models.

[20] Jianping Gou, Baosheng Yu, Stephen J. Maybank, and Dacheng Tao. 2021. Knowledge Distillation: A Survey. *International Journal of Computer Vision*, 129(6):1789–1819.

[21] Shuhao Gu, Mengdi Zhao, Bowen Zhang, Liangdong Wang, Jijie Li, and Guang Liu. 2024. ReTok: Replacing Tokenizer to Enhance Representation Efficiency in Large Language Model. ArXiv:2410.04335 [cs].

[22] Zhiwei He, Tian Liang, Jiahao Xu, Qiuzhi Liu, Xingyu Chen, Yue Wang, Linfeng Song, Dian Yu, Zhenwen Liang, Wenxuan Wang, Zhuosheng Zhang, Rui Wang, Zhaopeng Tu, Haitao Mi, and Dong Yu. 2025. DeepMath-103K: A Large-Scale, Challenging, Decontaminated, and Verifiable Mathematical Dataset for Advancing Reasoning. GSCC: 0000007 2025-06-25T15:17:46.851Z 1.40 arXiv:2504.11456 [cs].

[23] Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*.

[24] Brad Hilton, Kyle Corbitt, David Corbitt, Saumya Gandhi, Angky William, Bohdan Kovalenskyi, and Andie Jones. 2025. Art: Agent reinforcement trainer. `https://github.com/openpipe/art`.

[25] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the knowledge in a neural network.

[26] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the Knowledge in a Neural Network. ArXiv:1503.02531 [stat].

[27] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. ArXiv:2106.09685 [cs].

[28] Jingcheng Hu, Yinmin Zhang, Qi Han, Daxin Jiang, Xiangyu Zhang, and Heung-Yeung Shum. 2025. Open-reasoner-zero: An open source approach to scaling up reinforcement learning on the base model.

[29] Danqing Huang, Shuming Shi, Chin-Yew Lin, Jian Yin, and Wei-Ying Ma. 2016. How well do computers solve math word problems? large-scale dataset construction and evaluation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 887–896.

[30] Hugging Face. 2025. Open r1: A fully open reproduction of deepseek-r1.

[31] Vivek Iyer, Pinzhen Chen, and Alexandra Birch. 2023. Towards effective disambiguation for machine translation with large language models. *arXiv preprint arXiv:2309.11668*.

[32] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2704–2713, Salt Lake City, UT. IEEE.

[33] Kayvan Karim and Hani Ragab Hassen Hadj Batatia. 2025. Innovative tokenisation of structured data for LLM training. ArXiv:2508.01685 [cs].

[34] Amirhossein Kazemnejad, Milad Aghajohari, Alessandro Sordoni, Aaron Courville, and Siva Reddy. 2025. Nano aha! moment: Single file ”rl for llm” library. `https://github.com/McGill-NLP/nano-aha-moment`. GitHub repository.

[35] Taku Kudo and John Richardson. 2018. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing.

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

[37] Atte Laakso, Kai-Kristian Kemell, and Jukka K Nurminen. Ethical Issues in Large Language Models: A Systematic Literature Review.

[38] Nathan Lambert. 2025. *Reinforcement Learning from Human Feedback*. Online.

[39] Wang Ling, Dani Yogatama, Chris Dyer, and Phil Blunsom. 2017. Program induction by rationale generation: Learning to solve and explain algebraic word problems. *arXiv preprint arXiv:1705.04146*.

[40] Ruibo Liu, Jerry Wei, Fangyu Liu, Chenglei Si, Yanzhe Zhang, Jinmeng Rao, Steven Zheng, Daiyi Peng, Diyi Yang, Denny Zhou, and Andrew M. Dai. 2024. Best Practices and Lessons Learned on Synthetic Data. ArXiv:2404.07503 [cs].

[41] Tianqi Liu, Yao Zhao, Rishabh Joshi, Misha Khalman, Mohammad Saleh, Peter J. Liu, and Jialu Liu. 2024. Statistical Rejection Sampling Improves Preference Optimization. ArXiv:2309.06657 [cs].

[42] Llama.cpp. 2026. ggml-org/llama.cpp. Original-date: 2023-03-10T18:58:00Z.

[43] Pan Lu, Liang Qiu, Wenhao Yu, Sean Welleck, and Kai-Wei Chang. 2023. A survey of deep learning for mathematical reasoning. ArXiv:2212.10535 [cs].

[44] Alexandra Sasha Luccioni, Yacine Jernite, and Emma Strubell. 2024. Power Hungry Processing: Watts Driving the Cost of AI Deployment? In *The 2024 ACM Conference on Fairness Accountability and Transparency*, pages 85–99. ArXiv:2311.16863 [cs].

[45] Amir M. Mansourian, Rozhan Ahmadi, Masoud Ghafouri, Amir Mohammad Babaei, Elaheh Badali Golezani, Zeynab Yasamani Ghamchi, Vida Ramezanian, Alireza Taherian, Kimia Dinashi, Amirali Miri, and Shohreh Kasaei. 2025. A Comprehensive Survey on Knowledge Distillation. ArXiv:2503.12067 [cs].

[46] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: A Distributed Framework for Emerging AI Applications. ArXiv:1712.05889 [cs].

[47] OpenAI. 2026. Evaluating chain-of-thought monitorability.

[48] OpenPipe. 2026. OpenPipe/OpenPipe. Original-date: 2023-06-20T00:55:24Z.

[49] OpenStax. OpenStax. Open educational resources and textbooks (accessed 2025-12-23).

[50] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, et al. 2022. Training language models to follow instructions with human feedback. ArXiv:2203.02155 [cs].

[51] Jiayi Pan, Junjie Zhang, Xingyao Wang, Lifan Yuan, Hao Peng, and Alane Suhr. 2025. Tinyzero. https://github.com/Jiayi-Pan/TinyZero. Accessed: 2025-01-24.

[52] python-patterns.guide. 2026. Python Patterns Guide. Accessed: 2026-01-09.

[53] Changle Qu, Sunhao Dai, Xiaochi Wei, Hengyi Cai, Shuaiqiang Wang, Dawei Yin, Jun Xu, and Ji-Rong Wen. 2025. Tool Learning with Large Language Models: A Survey. *Frontiers of Computer Science*, 19(8). ArXiv:2405.17935 [cs].

[54] Qwen. 2025. Qwen/Qwen3-VL-32B-Instruct-FP8 · Hugging Face.

[55] Qwen, An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, Huan Lin, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jingren Zhou, Junyang Lin, et al. 2025. Qwen2.5 Technical Report. ArXiv:2412.15115 [cs].

[56] Runpod. 2026. Runpod | The cloud built for AI.

[57] Siddharth Samsi, Dan Zhao, Joseph McDonald, Baolin Li, Adam Michaleas, Michael Jones, William Bergeron, Jeremy Kepner, Devesh Tiwari, and Vijay Gadepally. 2023. From Words to Watts: Benchmarking the Energy Costs of Large Language Model Inference. ArXiv:2310.03003 [cs].

[58] David Saxton, Edward Grefenstette, Felix Hill, and Pushmeet Kohli. 2019. Analysing mathematical reasoning abilities of neural models. *arXiv preprint arXiv:1904.01557*.

[59] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. ArXiv:2302.04761 [cs].

[60] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms.

[61] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. ArXiv:1707.06347 [cs].

[62] Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. ArXiv:1508.07909 [cs].

[63] Javad SeraJ, Mohammad Mahdi Mohajeri, and Mohammad Javad Dousti. 2025. D^2LoRA: Data-Driven LoRA Initialization for Low Resource Tasks. ArXiv:2503.18089 [cs].

[64] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, et al. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

[65] Shaurya Sharthak, Vinayak Pahalwan, Adithya Kamath, and Adarsh Shirawalmath. 2025. Achieving Tokenizer Flexibility in Language Models through Heuristic Adaptation and Supertoken Learning. ArXiv:2505.09738 [cs] version: 1.

[66] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. 2024. Hybridflow: A flexible and efficient rlhf framework. *arXiv preprint arXiv:2409.19256*.

[67] Stack Exchange, Inc. Stack Exchange Data Dump. Public data dump for the Stack Exchange network (accessed 2025-12-23).

[68] stanford. 2023. Alpaca: A strong, replicable instruction-following model.

[69] L. Tunstall, L. von Werra, and T. Wolf. 2022. *Natural Language Processing with Transformers: Building Language Applications with Hugging Face*. O'Reilly Media.

[70] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention is all you need.

[71] vllm. 2024. [Bug]: lora base_model.model.lm_head.base_layer.weight is not supported · Issue #4186 · vllm-project/vllm.

[72] Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Tristan Thrush, Nathan Lambert, Shengyi Huang, Kashif Rasul, and Quentin Gallouédec. 2020. TRL: Transformers Reinforcement Learning.

[73] Changhan Wang, Kyunghyun Cho, and Jiatao Gu. 2019. Neural Machine Translation with Byte-Level Subwords. ArXiv:1909.03341 [cs].

[74] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2024. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345.

[75] Yiping Wang, Qing Yang, Zhiyuan Zeng, Liliang Ren, Liyuan Liu, Baolin Peng, Hao Cheng, Xuehai He, Kuan Wang, Jianfeng Gao, Weizhu Chen, Shuohang Wang, Simon Shaolei Du, and Yelong Shen. 2025. Reinforcement Learning for Reasoning in Large Language Models with One Training Example. ArXiv:2504.20571 [cs].

[76] Zhiruo Wang, Zhoujun Cheng, Hao Zhu, Daniel Fried, and Graham Neubig. 2024. What Are Tools Anyway? A Survey from the Language Model Perspective. ArXiv:2403.15452 [cs].

[77] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models.

[78] Lei Xu, Sirui Chen, Yuxuan Huang, and Chaochao Lu. 2025. Synthesis by Design: Controlled Data Generation via Structural Guidance. ArXiv:2506.07664 [cs].

[79] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, et al. 2025. Qwen3 Technical Report. ArXiv:2505.09388 [cs].

[80] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2024. τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains. ArXiv:2406.12045 [cs].

[81] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. ArXiv:2210.03629 [cs].

[82] Longhui Yu, Weisen Jiang, Han Shi, Jincheng Yu, Zhengying Liu, Yu Zhang, James T. Kwok, Zhenguo Li, Adrian Weller, and Weiyang Liu. 2024. MetaMath: Bootstrap Your Own Mathematical Questions for Large Language Models. ArXiv:2309.12284 [cs].

[83] Biao Zhang, Zhongtao Liu, Colin Cherry, and Orhan Firat. 2024. When scaling meets llm finetuning: The effect of data, model and finetuning method. *arXiv preprint arXiv:2402.17193*.

[84] Chong Zhang, Yue Deng, Xiang Lin, Bin Wang, Dianwen Ng, Hai Ye, Xingxuan Li, Yao Xiao, Zhanfeng Mo, Qi Zhang, and Lidong Bing. 2025. 100 Days After DeepSeek-R1: A Survey on Replication Studies and More Directions for Reasoning Language Models.

[85] Renrui Zhang, Jiaming Han, Chris Liu, Peng Gao, Aojun Zhou, Xiangfei Hu, Shilin Yan, Pan Lu, Hongsheng Li, and Yu Qiao. 2024. LLaMA-Adapter: Efficient Fine-tuning of Language Models with Zero-init Attention. ArXiv:2303.16199 [cs].

[86] Shengyu Zhang, Linfeng Dong, Xiaoya Li, Sen Zhang, Xiaofei Sun, Shuhe Wang, Jiwei Li, Runyi Hu, Tianwei Zhang, Fei Wu, and Guoyin Wang. 2025. Instruction Tuning for Large Language Models: A Survey. ArXiv:2308.10792 [cs].

[87] Ying Zhang, Tao Xiang, Timothy M. Hospedales, and Huchuan Lu. 2017. Deep mutual learning.

[88] Han Zhao, Haotian Wang, Yiping Peng, Sitong Zhao, Xiaoyu Tian, Shuaiting Chen, Yunjie Ji, and Xiangang Li. 2025. 1.4 Million Open-Source Distilled Reasoning Dataset to Empower Large Language Model Training.

[89] Chujie Zheng, Shixuan Liu, Mingze Li, Xiong-Hui Chen, Bowen Yu, Chang Gao, Kai Dang, Yuqiong Liu, Rui Men, An Yang, Jingren Zhou, and Junyang Lin. 2025. Group Sequence Policy Optimization. ArXiv:2507.18071 [cs].

[90] Álvaro Bartolomé Del Canto, Gabriel Martín Blázquez, Agustín Piqueres Lajarín, and Daniel Vila Suero. 2024. Distilabel: An ai feedback (aif) framework for building datasets with and for llms. `https://github.com/argilla-io/distilabel`.

Appendices

A.1 Released Artifacts

A Deliverables

Artifact	Phase	Contents / traces
Model checkpoints		
SFT Checkpoint	SFT	Standalone SFT checkpoint (wandb run).
SFT LoRA Checkpoint	SFT	Model with LoRA adapters (wandb run).
GRPO Checkpoint	GRPO	GRPO with LoRA adapters (wandb run).
Datasets		
Base Dataset	Generation	Verified synthetic linear-algebra problems.
Raw Distilled Dataset	Distillation	Generated multi-turn traces before filtering.
Distilled Dataset	Distillation	Dataset used during SFT.
Distilled Dataset Failures	Distillation	Rejected distillation traces.
SFT Dataset	SFT	Derived from the distillation phase.
GRPO Dataset	GRPO	Derived from the base dataset.
Evaluation		
Evaluation Results	Evaluation	Aggregated evaluation metrics and runs.
Failure Breakdown	Evaluation	Failure analysis reproducibility data.
Traces		
Base Model Traces	Base Model	Base model traces (prior to training).

Table A.1: Project deliverables and reproducibility links. All generated traces are available from each individual wandb run (see Evaluation tab).

A.2 Generation

A Linear Algebra Primitives

Definition A.1 (Matrix transpose)**.** The transpose operation flips a matrix over its main diagonal, switching the row and column indices of the matrix. Formally, for a matrix A, the element at position (i, j) in A^T is the element at (j, i) in A.

Example (Transpose of a 2×2 matrix)**.**

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \implies A^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

Definition A.2 (Matrix trace)**.** The trace is defined only for square matrices ($n \times n$). It is the sum of the elements on the main diagonal (from the upper left to the lower right). This metric is invariant under cyclic permutations, making it a critical property for verifying similarity transformations.

Example (Trace of a 2×2 matrix)**.**

$$A = \begin{bmatrix} \mathbf{5} & 2 \\ 1 & \mathbf{3} \end{bmatrix} \implies \mathrm{tr}(A) = 5 + 3 = 8$$

Definition A.3 (Determinant)**.** The determinant is a scalar value that characterizes the scaling factor of the linear transformation described by the matrix. Crucially, it acts as a solvability check: if $\det(A) = 0$, the matrix is singular (non-invertible).

Example (Determinant of a 2×2 matrix)**.**

$$\det \begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc$$

Definition A.4 (Frobenius norm)**.** This operation measures the size (magnitude) of a matrix in Euclidean space. It is calculated as the square root of the sum of the absolute squares of its elements. In our dataset, this operation serves as a test of the model's ability to handle floating-point accumulation.

Example (Frobenius norm of a 2×2 matrix)**.**

$$A = \begin{bmatrix} 2 & -1 \\ 0 & 2 \end{bmatrix} \implies ||A||_F = \sqrt{2^2 + (-1)^2 + 0^2 + 2^2} = \sqrt{9} = 3$$

Definition A.5 (Matrix rank)**.** The rank represents the maximum number of linearly independent rows (or columns) in the matrix. It is a measure of the information content of the matrix. A system of linear equations $Ax = b$ has a unique solution only if A is full rank.

Example (A rank-1 matrix)**.**

$$A = \begin{bmatrix} 1 & 0 \\ 2 & 0 \end{bmatrix}$$

Here, the second row is simply 2× the first row. They are dependent. Therefore, $\mathrm{rank}(A) = 1$.

Definition A.6 (Matrix cofactor)**.** The cofactor is an intermediate step required for calculating the matrix inverse. For a given element a_{ij}, the cofactor C_{ij} is the signed determinant of the sub-matrix left behind when row i and column j are removed.

Example (Computing a cofactor)**.** For $A = \begin{bmatrix} 1 & 4 \\ 3 & 5 \end{bmatrix}$, to find C_{12} (row 1, col 2):

1. Remove row 1 and column 2. Remaining element: 3.
2. Apply sign $(-1)^{1+2} = -1$.
3. Result: $C_{12} = -3$.

B Problem-Type Taxonomy

Tool-call chain	# tool-calls
`determinant`	1
`frobenius_norm`	1
`matrix_cofactor`	1
`matrix_rank`	1
`matrix_trace`	1
`matrix_transpose`	1
`matrix_cofactor` → `matrix_rank`	2
`matrix_cofactor` → `matrix_trace`	2
`matrix_transpose` → `determinant`	2
`matrix_transpose` → `frobenius_norm`	2
`matrix_cofactor` → `matrix_transpose` → `matrix_trace`	3
`matrix_transpose` → `matrix_cofactor` → `frobenius_norm`	3
`matrix_transpose` → `matrix_cofactor` → `matrix_rank`	3

Table B.1: Problem-type taxonomy for dataset generation. Each type is defined by an ordered tool-call chain, and difficulty is the number of tool calls required (1/2/3).

C Representative Generated Examples

In this section, we list the dataset schema and a few query strings as representative dataset entries.

Single-step example

Problem type. `one_matrix_rank`.
Query.

```
Find rank(A) for A = [[-44, -122, 88], [-108, 83, -43], [87, -84, -92]].
```

Ground truth. 3.
Stepwise ground truth.

```
[{"matrix_rank": 3}]
```

Two-step example

Problem type. `two_transpose_determinant`.
Query.

```
Step 1: find A^T for A = [[4, 2, 2], [-3, 4, -2], [3, -2, 3]].
Step 2: evaluate the determinant of matrix B, where B = the result from step 1.
```

Ground truth. 26.0.
Stepwise ground truth.

```
[
  {"matrix_transpose": [[4, -3, 3], [2, 4, -2], [2, -2, 3]]},
  {"determinant": 26.0}
]
```

Three-step example

Problem type. `three_cofactor_transpose_trace`.
Query.

```
Step 1: what is the matrix of cofactors for A = [[-132, -116], [211, -239]]?
Step 2: find B^T for B = the result from step 1.
Step 3: determine the trace of matrix C = the result from step 2.
```

Ground truth. -371.0.
Stepwise ground truth.

```
[
  {"matrix_cofactor": [[-239, -211], [116, -132]]},
  {"matrix_transpose": [[-239, 116], [-211, -132]]},
  {"matrix_trace": -371.0}
]
```

D Generation Algorithm

```
procedure GenerateQuestionSingleStep(task, entropyBounds, constraints)
    E ← SampleEntropy(entropyBounds)
    context ← ProblemContext(entropy=E)
    instance ← GenerateMathContent(task, context, constraints)
    question ← FormatQuestion(instance)
    answer ← FormatAnswer(instance)
    Verify(instance)                                    ▷ reject if verification fails
    trace ← context.stepwiseTrace
    return (question, answer, trace)
end procedure
```

Algorithm D.1. This algorithm generates a verified single-step linear algebra instance.

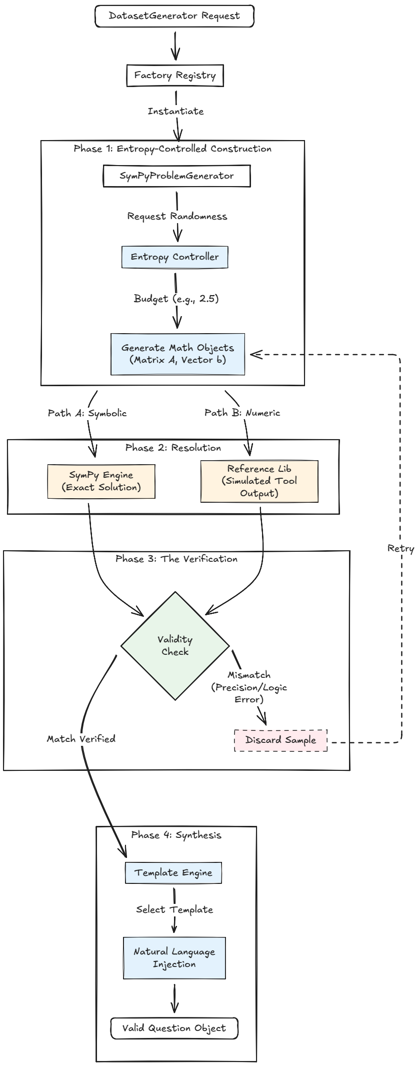


Figure D.1: Generation pipeline for synthetic problems. Each instance is accepted only if independent executions agree under deterministic replay.

E Generation Schematics

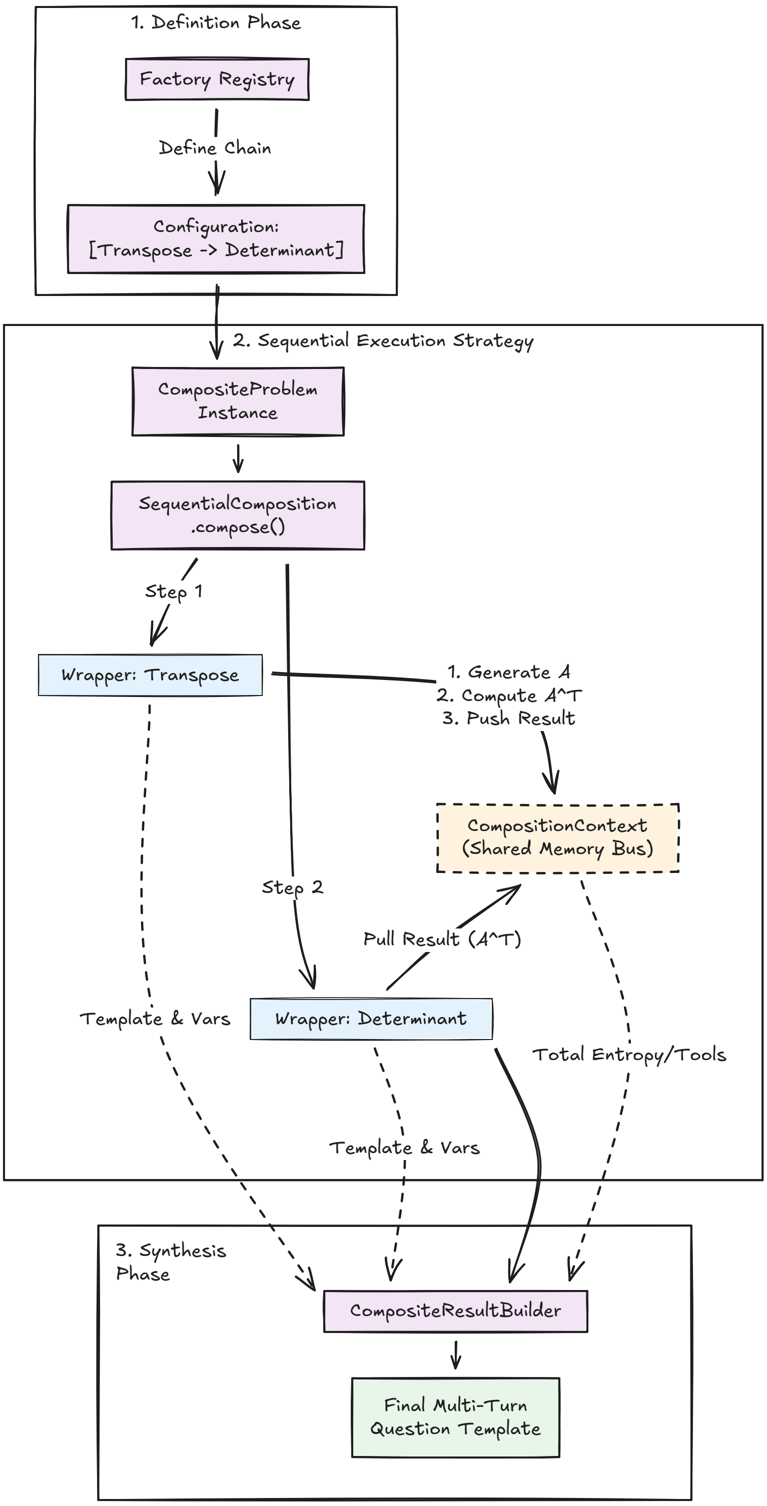


Figure E.1: Problem construction for multi-step questions. Later steps consume exact outputs of earlier tool calls via a shared context.

A.3 Distillation

A System Prompt

Role	Message content
system	You are an expert in composing functions. You are given a math problem from a user and a set of possible functions. Based on the question, you will need to make one function/tool call at a time to complete the task. For each step: 1. Start: Begin each turn with a brief plan inside <think> </think> tags. The plan should identify WHICH tool to call and WHY, not perform calculations. Focus on: (a) what information you need, (b) which tool provides it, (c) how it connects to the next step. 2. Tool Usage: Always use tools for computation--never perform manual calculations, arithmetic, or matrix operations in your reasoning. When a step requires computation, emit exactly ONE tool call by writing a JSON command inside <tool_call> </tool_call> tags with name and arguments keys. Example: <tool_call> {{"name": "matrix_transpose", "arguments": {{"matrix": [[1, 2], [3, 4]]}}}} </tool_call> Tools expect specific JSON input formats. Follow the examples carefully. Do not make up tools or arguments that aren't listed. 3. Tool Response: After you use a tool, you will see the tool output inside <tool_response> <tool_response> tags from the system. Use this result to inform your next step (either another tool call or, if complete, your final answer). 4. Error Recovery: If a tool result seems unexpected, you may call a different tool to cross-check, but do not manually verify or recompute results. The tools are your computational engine--your role is orchestration, not calculation. 5. Structure: The <think> </think> block must contain only planning--no nested tags, no arithmetic. Each turn must contain exactly one thinking block followed by either an answer block OR a tool call block (but never both). 6. Mutual Exclusion: Never output <answer> and <tool_call> in the same turn. Only output <answer> after receiving the final <tool_response> and when no further tool calls are necessary. 7. Final Answer: Your final answer must be taken directly from the result in the final <tool_response> </tool_response> tags -- not from intermediate tool calls. When the task is fully solved, output the final answer inside the <answer> </answer> block. The answer must contain ONLY the mathematical result (numeric, vector, or matrix) from the final tool response in its simplest form, with no descriptive text or intermediate values.

Table A.1: System prompt defining the strict tool-use contract used during distillation and downstream training.

B Tool-Use Trace Examples

Role	Message content
Example A (single-step; train[1500]).	
system	[System prompt excerpt: Use <think> then either <tool_call> or <answer>. Tool calls must be JSON with a declared tool name and arguments. Avoid manual computation; ground the final <answer> in the last tool output.]
user	Evaluate the determinant of matrix A, where A = [[-5, -2, -4], [-5, 3, -2], [-5, -4, 3]].
assistant	<think>Okay, I need to find the determinant of matrix A. The matrix is 3x3, so I can use the determinant tool directly. The tool takes the matrix as input and returns the determinant. I don't need to do any manual calculations here. Let me call the determinant function with the given matrix.</think>

tool call	name=determinant
	arguments={"matrix": [[-5, -2, -4], [-5, 3, -2], [-5, -4, 3]]}

tool	-195.0
assistant	<think>Okay, the user asked for the determinant of matrix A. I used the determinant tool with the provided matrix, and the result was -195.0. Since the determinant is calculated directly by the tool, and there are no further steps needed, the answer should be -195.0. I just need to confirm that the tool's output is correct and then present it as the final answer.</think> <answer>-195.0</answer>
Example B (three-step tool use; train[3522]).	
user	Step 1: evaluate the transpose of matrix A = [[545, 170], [677, -526]]. Step 2: what is the matrix of cofactors for B = the result from step 1? Step 3: find the rank of matrix C = the result from step 2.
assistant	<think>Okay, let's tackle this step by step. First, the user wants the transpose of matrix A. Matrix A is [[545, 170], [677, -526]]. The transpose swaps rows and columns, so I'll use the matrix_transpose tool for step 1. Once I get the transpose (let's call it matrix B), step 2 requires finding its cofactor matrix. The matrix_cofactor tool is the right choice here. After obtaining the cofactor matrix (matrix C), step 3 is to determine its rank using the matrix_rank tool. Each step depends on the previous result, so I'll need to perform each tool call sequentially, using the output of one as the input for the next.</think>

tool call	name=matrix_transpose
	arguments={"matrix": [[545, 170], [677, -526]]}

tool	[[545, 677], [170, -526]]
assistant	<think>Okay, so the first step was to find the transpose of matrix A. The tool gave me the transposed matrix B as [[545, 677], [170, -526]]. Now, I need to find the cofactor matrix of B for step 2. The matrix_cofactor tool is the right choice here. The cofactor matrix involves calculating minors and applying the checkerboard of signs. Since B is a 2x2 matrix, the cofactor matrix should be straightforward. Let me call the matrix_cofactor tool with matrix B as the input. Once I get that, I'll move on to step 3, which is finding the rank of the resulting cofactor matrix.</think>

tool call	name=matrix_cofactor
	arguments={"matrix": [[545, 677], [170, -526]]}

tool	[[-526, -170], [-677, 545]]
assistant	<think>Okay, so the cofactor matrix of B is [[-526, -170], [-677, 545]]. Now, step 3 requires finding the rank of this matrix. The matrix_rank tool will determine the number of linearly independent rows or columns. Since this is a 2x2 matrix, if the determinant is non-zero, the rank should be 2. If the determinant is zero, the rank would be 1. Let me use the matrix_rank tool on this cofactor matrix to get the result.</think>

tool call	name=matrix_rank
	arguments={"matrix": [[-526, -170], [-677, 545]]}

tool	2
assistant	<think>Okay, let me go through the steps again to make sure everything is correct. First, I transposed matrix A, which gave me matrix B. Then, I found the cofactor matrix of B, resulting in matrix C. Finally, I determined the rank of matrix C. The tool returned a rank of 2. Since matrix C is a 2x2 matrix and its rank is 2, that means it's full rank, so the answer should be 2. I think that's all. No mistakes in the steps, and the tools were used correctly each time.</think> <answer>2</answer>

Table B.1: Traces produced during the distillation process, illustrating stepwise planning, tool chaining and end-of-trace self-verification. For traces of failed examples see this link.

A.4 Reinforcement Learning

A Reward Function

```
async def calculate_reward(self) -> int:
    """Composite reward that combines several components:

    - Correctness (primary signal)
    - Formatting of thoughts/answers
    - Tool-call success (to ensure executable tool-calls)
    - Efficiency (penalizes over/under-using tools)
    """
    # Handle structurally invalid trajectories
    if not self.actions:
        return -1.0

    tool_calls = self.actions[:-1]
    answer = self.actions[-1]

    # No tool calls: treat as maximally lazy
    if len(tool_calls) == 0:
        return -0.3

    # Final step must be a respond action
    if answer.name != RESPOND_ACTION_NAME:
        return -1.0

    # Compute individual components
    correctness = self.correctness_reward()
    format_score = self.format_reward()
    tool_success = self.tool_success_reward()
    efficiency_penalty = self.efficiency_penalty()

    # Apply weights to each component
    total_reward (
          1.0 * correctness
        + 0.1 * format_score
        + 0.1 * tool_success
        - 0.1 * efficiency_penalty
    ) / 1.2

    return total_reward
```

B GRPO/SFT Side-by-Side Comparison

Role	Message content
Matched Example (transpose → determinant; 2 tool calls).	
user	Step 1: what is the transpose of A = [[2, 4, -2], [-2, -4, -3], [4, 4, -2]]? Step 2: for B = the result from step 1, compute det(B).
SFT (task_index 380).	
assistant	<think>Okay, let's tackle this problem step by step. The user has two steps: first, find the transpose of matrix A, and then compute the determinant of that transposed matrix. Starting with Step 1: The transpose of a matrix involves swapping its rows and columns. So, for matrix A, which is a 3x3 matrix, the transpose should also be a 3x3 matrix where the element at position (i,j) in the original matrix becomes (j,i) in the transposed matrix. To do this, I need to use the matrix_transpose tool. The input matrix A is [[2, 4, -2], [-2, -4, -3], [4, 4, -2]]. Applying the transpose tool should give me the correct transposed matrix. Once I have the transposed matrix B, Step 2 requires calculating its determinant. The determinant of a 3x3 matrix can be computed using the determinant tool. However, I need to make sure that the matrix is square, which it is since the transpose of a square matrix is also square. So, after getting the transpose, I'll pass it to the determinant function to get the result. I should proceed by first calling the matrix_transpose tool with matrix A. Then, take the output and use it as the input for the determinant tool. Let's start with the first tool call.</think>

tool call	name=matrix_transpose
	arguments="{\"matrix\": [[2, 4, -2], [-2, -4, -3], [4, 4, -2]]}"

tool	[[2, -2, 4], [4, -4, 4], [-2, -3, -2]]
assistant	<think>Okay, so the first step was to find the transpose of matrix A. The user provided matrix A as [[2, 4, -2], [-2, -4, -3], [4, 4, -2]]. Using the matrix_transpose tool, the result is [[2, -2, 4], [4, -4, 4], [-2, -3, -2]]. That makes sense because transposing swaps rows and columns. Now, the next step is to compute the determinant of this transposed matrix, which is matrix B. To find the determinant of a 3x3 matrix, I need to use the determinant tool. The determinant of a matrix is calculated by taking the sum of each element in the first row multiplied by the determinant of the 2x2 matrix that remains after removing that row and column. But instead of doing the manual calculation, I should use the determinant function provided. The input matrix B is [[2, -2, 4], [4, -4, 4], [-2, -3, -2]]. I need to pass this matrix to the determinant tool. Let me make sure the matrix is correctly formatted as a list of rows. The tool should handle the calculation, so I just need to call it with the transposed matrix as the argument. The result will be the determinant, which should be a single number. Once I get that, I can present it as the final answer for step 2.</think>

tool call	name=determinant
	arguments="{\"matrix\": [[2, -2, 4], [4, -4, 4], [-2, -3, -2]]}"

tool	-40.0
assistant	<think>Okay, let's see. The user asked for two steps. First, find the transpose of matrix A, which they did using the matrix_transpose tool. The result was [[2, -2, 4], [4, -4, 4], [-2, -3, -2]]. Then, step two is to compute the determinant of that transposed matrix. So, I need to use the determinant tool on the transposed matrix. The determinant of a 3x3 matrix can be calculated using the rule of Sarrus or cofactor expansion. But since the tool is available, I should just call it with the transposed matrix as the argument. The tool response gave -40.0. That's the determinant. Since the user's final question was for the determinant of the transposed matrix, this is the answer they're looking for. No more steps needed.</think> <answer>-40.0</answer>
GRPO (task_index 380).	
assistant	<think>Okay, let's tackle this problem step by step. The user has two steps: first, find the transpose of matrix A, and then compute the determinant of that transposed matrix. Starting with Step 1: The transpose of a matrix involves swapping its rows and columns. So, for matrix A, which is a 3x3 matrix, the transpose should also be a 3x3 matrix where the element at position (i,j) in the original matrix becomes (j,i) in the transposed matrix. To do this, I need to use the matrix_transpose tool. The input matrix A is [[2, 4, -2], [-2, -4, -3], [4, 4, -2]]. Applying the transpose tool should give me the correct transposed matrix. Once I have the transposed matrix B, Step 2 requires calculating its determinant. The determinant of a 3x3 matrix can be computed using the determinant tool. However, since the transpose of a matrix doesn't change its determinant, the determinant of B should be the same as the determinant of A. But I need to confirm this by actually computing it. So, after getting the transpose, I'll use the determinant tool on the resulting matrix B to get the final answer. I should make sure to use the tools correctly. First, call matrix_transpose with matrix A, then take the result and pass it to determinant. Let's start with the first tool call.</think>

Role	Message content

tool call	name=matrix_transpose
	arguments="{\"matrix\": [[2, 4, -2], [-2, -4, -3], [4, 4, -2]]}"

tool	[[2, -2, 4], [4, -4, 4], [-2, -3, -2]]
assistant	<think>Okay, so the first step was to find the transpose of matrix A. The user provided matrix A as [[2, 4, -2], [-2, -4, -3], [4, 4, -2]]. Using the matrix_transpose tool, the result is [[2, -2, 4], [4, -4, 4], [-2, -3, -2]]. That makes sense because transposing swaps the rows and columns. Now, the next step is to compute the determinant of this transposed matrix, which is matrix B. Since the determinant of a matrix and its transpose are the same, I should use the determinant tool on matrix B. The determinant of a 3x3 matrix can be calculated using the rule of Sarrus or cofactor expansion. However, the tool will handle the computation, so I just need to pass the transposed matrix to the determinant function. Let me make sure the matrix is correctly formatted as a list of lists. The transposed matrix is [[2, -2, 4], [4, -4, 4], [-2, -3, -2]]. I'll call the determinant tool with this matrix as the argument.</think>

tool call	name=determinant
	arguments="{\"matrix\": [[2, -2, 4], [4, -4, 4], [-2, -3, -2]]}"

tool	-40.0
assistant	<think>Okay, let's see. The user asked for two steps. First, find the transpose of matrix A, which they did using the matrix_transpose tool. The result was [[2, -2, 4], [4, -4, 4], [-2, -3, -2]]. Then, step two is to compute the determinant of that transposed matrix. So, I need to use the determinant tool on the transposed matrix. The determinant of a 3x3 matrix can be calculated using the rule of Sarrus or cofactor expansion. But since the tool is available, I should just call it with the transposed matrix as the argument. The tool returned -40.0. That makes sense because the determinant of a matrix and its transpose are the same. So the final answer should be -40.0. I need to make sure there's no mistake in the calculation, but the tool's result is straightforward. The answer is correct.</think> <answer>-40.0</answer>

Table B.1: Two matched evaluation problems generated during SFT and GRPO.

Role	Message content
Matched Example (cofactor → rank; 2 tool calls).	
user	Step 1: find the cofactor matrix for A = [[7, -8, 9], [-6, 6, -8], [5, 2, -15]]. Step 2: find rank(B) for B = the result from step 1.
SFT (task_index 18).	
assistant	<think>Okay, let's tackle this problem step by step. The user wants me to first find the cofactor matrix of matrix A, and then determine the rank of that cofactor matrix (which is matrix B in step 2). Starting with step 1: finding the cofactor matrix of A. The matrix A is a 3x3 matrix. The cofactor matrix involves calculating the determinant of each minor matrix and applying the appropriate sign based on the position. Since manually calculating this can be error-prone, I should use the matrix_cofactor tool. This tool takes the matrix as input and returns the cofactor matrix. So I'll call that tool with matrix A. Once I get the cofactor matrix from the tool, step 2 requires finding its rank. The rank of a matrix is the maximum number of linearly independent rows or columns. For a 3x3 matrix, if the determinant is non-zero, the rank is 3. If the determinant is zero, the rank is less than 3. However, since the cofactor matrix might have a determinant of zero, I need to use the matrix_rank tool on the result from step 1. This tool will compute the rank directly. I need to make sure that after getting the cofactor matrix, I pass it correctly to the matrix_rank tool. Let's start with the first tool call.</think>

tool call	name=matrix_cofactor
	arguments="{\"matrix\": [[7, -8, 9], [-6, 6, -8], [5, 2, -15]]}"

tool	[[-74, -130, -42], [-102, -150, -54], [10, 2, -6]]
assistant	<think>Okay, so the first step was to find the cofactor matrix of A, and the tool gave me the result [[-74, -130, -42], [-102, -150, -54], [10, 2, -6]]. Now, step 2 is to find the rank of this cofactor matrix, which is matrix B. To find the rank, I need to determine the number of linearly independent rows or columns. Since it's a 3x3 matrix, if the determinant is non-zero, the rank should be 3. But if the determinant is zero, the rank would be less. However, calculating the determinant manually might be error-prone, so I should use the matrix_rank tool. The tool will handle the computation, so I just need to pass the cofactor matrix to it. Let me make sure the matrix is correctly formatted as a list of rows. The tool should take the matrix as input and return the rank. I'll call the matrix_rank function with the given cofactor matrix.</think>

tool call	name=matrix_rank
	arguments="{\"matrix\": [[-74, -130, -42], [-102, -150, -54], [10, 2, -6]]}"

tool	3
assistant	<think>Okay, let's see. The user asked for two steps. First, find the cofactor matrix of A, which they did using the matrix_cofactor tool. The result was a 3x3 matrix. Then, step two is to find the rank of that cofactor matrix. The matrix_rank tool was used, and it returned 3. Since the rank is 3, that means the matrix is full rank, which makes sense for a 3x3 matrix. The determinant of the cofactor matrix must be non-zero, so the rank is indeed 3. No further steps needed here. The final answer should be 3.</think> <answer>3</answer>
GRPO (task_index 18).	
assistant	<think>Okay, let's tackle this problem step by step. The user wants me to first find the cofactor matrix of matrix A, and then determine the rank of that cofactor matrix (which is matrix B in step 2). Starting with step 1: finding the cofactor matrix of A. The matrix A is a 3x3 matrix. The cofactor matrix involves calculating the determinant of each minor matrix and applying the appropriate sign based on the position. Since manually calculating this can be error-prone, I should use the matrix_cofactor tool provided. This tool takes the matrix as input and returns the cofactor matrix. So I'll call that tool with matrix A. Once I get the cofactor matrix from step 1, step 2 requires finding its rank. The rank of a matrix is the maximum number of linearly independent rows or columns. The matrix_rank tool can compute this. I'll use the result from step 1 as the input for this tool. I need to make sure that after getting the cofactor matrix, I pass it correctly to the matrix_rank tool. Let's start with the first tool call.</think>

tool call	name=matrix_cofactor
	arguments="{\"matrix\": [[7, -8, 9], [-6, 6, -8], [5, 2, -15]]}"

tool	[[-74, -130, -42], [-102, -150, -54], [10, 2, -6]]

Role	Message content
assistant	<think>Okay, so the first step was to find the cofactor matrix of A, and the tool gave me the result [[-74, -130, -42], [-102, -150, -54], [10, 2, -6]]. Now, step 2 is to find the rank of this cofactor matrix, which is matrix B. To find the rank, I need to determine the number of linearly independent rows or columns. The matrix_rank tool should handle this. I'll input the cofactor matrix into that tool. Let me make sure the matrix is correctly formatted as a list of rows. The tool's parameters require a matrix, so I'll pass the result from the first step. Once I get the rank from the tool, that will be the final answer for step 2.</think>

tool call	name=matrix_rank
	arguments="{\"matrix\": [[-74, -130, -42], [-102, -150, -54], [10, 2, -6]]}"

tool	3
assistant	<think>Okay, let's see. The user asked for two steps. First, find the cofactor matrix of A, which I did using the matrix_cofactor tool. The result was a 3x3 matrix. Then, step two is to find the rank of that cofactor matrix. I used the matrix_rank tool on the cofactor matrix and got a rank of 3. Since the matrix is 3x3 and the rank is 3, that means it's full rank. So the final answer should be 3. I need to make sure there's no mistake in the calculations, but the tools did the work, so the answer is correct.</think> <answer>3</answer>

Table B.2: Two matched evaluation problems generated during SFT and GRPO.

C Malformed Evaluation Examples

Role	**Message content (verbatim)**
Exclusive failure reasons (priority order).	
Forced stop (SFT eval, task_index 356, pass 1). **Failure reason:** Forced stop (Reached maximum allowed tool calls).	
user	`What is the transpose of A = [[-83, -427, 704], [-408, 181, 206], [-256, -260, -358]]?`

assistant	`<think>Okay, I need to find the transpose of matrix A. The user provided the matrix as a 3x3 array. The transpose of a matrix is obtained by swapping its rows with columns. So, the first row becomes the first column, the second row becomes the second column, and the third row becomes the third column.`
	`Looking at the tools available, there's a matrix_transpose function that takes a matrix and returns its transpose. That's exactly what I need here. I don't need to calculate anything manually; I can just call the matrix_transpose tool with the given matrix as the argument. The tool should handle the swapping correctly. Let me make sure the input is in the correct format. The matrix is already provided as a list of rows, which matches the required input type. So, I'll proceed to call the matrix_transpose tool with the given matrix.</think>`

tool call	`name=matrix_transpose`
	`arguments="{\"matrix\": [[-83, -427, 704], [-408, 181, 206], [-256, -260, -358]]}"`

tool	`[[-83, -408, -256], [-427, 181, -260], [704, 206, -358]]`

assistant	`<think>Okay, so I need to find the transpose of matrix A. The user provided the matrix as a 3x3 array. The transpose of a matrix is obtained by swapping its rows with columns. So, the first row becomes the first column, the second row becomes the second column, and the third row becomes the third column.`
	`Looking at the tools available, there's a matrix_transpose function that takes a matrix and returns its transpose. That's exactly what I need here. I don't need to calculate anything manually; I can just call the matrix_transpose tool with the given matrix as the argument. The tool should handle the swapping correctly. Let me make sure the input is in the correct format. The matrix is already provided as a list of rows, which matches the required input type. So, I'll proceed to call the matrix_transpose tool with the given matrix.</think>`

tool call	`name=matrix_transpose`
	`arguments="{\"matrix\": [[-83, -427, 704], [-408, 181, 206], [-256, -260, -358]]}"`

tool	`[[-83, -408, -256], [-427, 181, -260], [704, 206, -358]]`

assistant	`<think>Okay, so I need to find the transpose of matrix A. The user provided the matrix as a 3x3 array. The transpose of a matrix is obtained by swapping its rows with columns. So, the first row becomes the first column, the second row becomes the second column, and the third row becomes the third column.`
	`Looking at the tools available, there's a matrix_transpose function that takes a matrix and returns its transpose. That's exactly what I need here. I don't need to calculate anything manually; I can just call the matrix_transpose tool with the given matrix as the argument. The tool should handle the swapping correctly. Let me make sure the input is in the correct format. The matrix is already provided as a list of rows, which matches the required input type. So, I'll proceed to call the matrix_transpose tool with the given matrix.</think>`

tool call	`name=matrix_transpose`
	`arguments="{\"matrix\": [[-83, -427, 704], [-408, 181, 206], [-256, -260, -358]]}"`

tool	`[[-83, -408, -256], [-427, 181, -260], [704, 206, -358]]`

assistant	`<think>Okay, so I need to find the transpose of matrix A. The user provided the matrix as a 3x3 array. The transpose of a matrix is obtained by swapping its rows with columns. So, the first row becomes the first column, the second row becomes the second column, and the third row becomes the third column.`
	`Looking at the tools available, there's a matrix_transpose function that takes a matrix and returns its transpose. That's exactly what I need here. I don't need to calculate anything manually; I can just call the matrix_transpose tool with the given matrix as the argument. The tool should handle the swapping correctly. Let me make sure the input is in the correct format. The matrix is already provided as a list of rows, which matches the required input type. So, I'll proceed to call the matrix_transpose tool with the given matrix.</think>`

tool call	`name=matrix_transpose`
	`arguments="{\"matrix\": [[-83, -427, 704], [-408, 181, 206], [-256, -260, -358]]}"`

tool	`[[-83, -408, -256], [-427, 181, -260], [704, 206, -358]]`

Role	Message content (verbatim)
assistant	<think>Okay, so I need to find the transpose of matrix A. The user provided the matrix as a 3x3 array. The transpose of a matrix is obtained by swapping its rows with columns. So, the first row becomes the first column, the second row becomes the second column, and the third row becomes the third column. Looking at the tools available, there's a matrix_transpose function that takes a matrix and returns its transpose. That's exactly what I need here. I don't need to calculate anything manually; I can just call the matrix_transpose tool with the given matrix as the argument. The tool should handle the swapping correctly. Let me make sure the input is in the correct format. The matrix is already provided as a list of rows, which matches the required input type. So, I'll proceed to call the matrix_transpose tool with the given matrix.</think>

tool call	name=matrix_transpose
	arguments="{\"matrix\": [[-83, -427, 704], [-408, 181, 206], [-256, -260, -358]]}"

tool	[[-83, -408, -256], [-427, 181, -260], [704, 206, -358]]
GRPO success example (task_index 356, pass 2).	
user	What is the transpose of A = [[-83, -427, 704], [-408, 181, 206], [-256, -260, -358]]?
assistant	<think>Okay, I need to find the transpose of matrix A. The transpose of a matrix is formed by swapping its rows with columns. So, the element in the first row and second column becomes the element in the second row and first column, and so on. Looking at the given matrix A: Row 1: [-83, -427, 704] Row 2: [-408, 181, 206] Row 3: [-256, -260, -358] To transpose this, the first column becomes the first row, the second column becomes the second row, and the third column becomes the third row. So the transposed matrix should look like: First row: [-83, -408, -256] Second row: [-427, 181, -260] Third row: [704, 206, -358] But instead of doing this manually, I should use the matrix_transpose tool provided. The tool takes the matrix as input and returns the transposed matrix. So I'll call the matrix_transpose function with matrix A as the argument. The result from the tool will be the correct transpose, which I can then present as the final answer.</think>

tool call	name=matrix_transpose
	arguments="{\"matrix\": [[-83, -427, 704], [-408, 181, 206], [-256, -260, -358]]}"

tool	[[-83, -408, -256], [-427, 181, -260], [704, 206, -358]]
assistant	<think>Okay, so I need to find the transpose of matrix A. The user provided the matrix as [[-83, -427, 704], [-408, 181, 206], [-256, -260, -358]]. To transpose a matrix, I need to swap its rows and columns. First, let me recall what a transpose is. The transpose of a matrix is obtained by turning rows into columns and columns into rows. So, the element in the first row and second column becomes the element in the second row and first column, and so on. Looking at the original matrix: - The first row is [-83, -427, 704]. - The second row is [-408, 181, 206]. - The third row is [-256, -260, -358]. To transpose this, the first column becomes the first row, the second column becomes the second row, and the third column becomes the third row. So the transposed matrix should look like: First row: [-83, -408, -256] Second row: [-427, 181, -260] Third row: [704, 206, -358] Wait, but the user already provided the transposed matrix as [[-83, -408, -256], [-427, 181, -260], [704, 206, -358]]. Let me check if this matches the expected result. Yes, each element is correctly placed in the transposed position. For example, the element at [0][1] in the original matrix is -427, which is now at [1][0] in the transposed matrix. Similarly, the element at [2][2] in the original is -358, which is at [2][2] in the transposed matrix. Since the tool's response matches the expected result, the transpose is correct. Therefore, the final answer should be the transposed matrix provided by the tool.</think> <answer>[[-83, -408, -256], [-427, 181, -260], [704, 206, -358]]</answer>

Valid trajectory = 0 (SFT eval, task_index 110, pass 1).

Failure reason: Valid trajectory = 0 (Trace invalid after parsing – e.g. truncated tags).

user	Step 1: find the cofactor matrix for A = [[100, 99], [47, -7]]. Step 2: calculate the transpose of matrix B = the result from step 1. Step 3: what is the trace of C = the result from step 2?

Role	Message content (verbatim)
assistant	<think>Okay, let's tackle this problem step by step. First, I need to find the cofactor matrix of matrix A. The matrix given is [[100, 99], [47, -7]]. The cofactor matrix involves calculating the minors and applying the appropriate signs based on the position. For a 2x2 matrix, the cofactor matrix can be found by swapping the diagonal elements and changing the signs of the off-diagonal elements. So, the cofactor of element a11 (100) is (-7), a12 (99) becomes -47, a21 (47) becomes -99, and a22 (-7) remains 100. Wait, no, the cofactor for each element is (-1)^(i+j) multiplied by the minor. For a21, the minor is the determinant of the submatrix obtained by removing row 2 and column 1, which is just -7. So the cofactor is (-1)^(2+1) * (-7) = -(-7) = 7. Similarly, for a12, the minor is 47, so the cofactor is (-1)^(1+2) * 47 = -47. Wait, no, the cofactor for a12 is (-1)^(1+2) * (-7) = -(-7) = 7. Wait, no, the cofactor for a12 is (-1)^(1+2) * (-7) = -(-7) = 7. Wait, no, the cofactor for a12 is (-1)^(1+2) * (-7) = -(-7) = 7. Wait, no, the cofactor for a12 is (-1)^(1+2) * (-7) = -(-7) = 7. Wait, no, the cofactor for a12 is (-1)^(1+2) * (-7) = -(-7) = 7. Wait, no, the cofactor for a12 is (-1)^(1+2) * (-7) = -(-7) = 7. Wait, no, the cofactor for a12 is (-1)^(1+2) * (-7) = -(-7) = 7. Wait, no, the cofactor for a12 is (-1)^(1+2) * (-7) = -(-7) = 7. Wait, no, the cofactor for a12 is (-1)^(1+2) * (-7) = -(-7) = 7. Wait, no, the cofactor for a12 is (-1)^(1+2) * (-7) = -(-7) = 7. Wait, no, the cofactor for a12 is (-1)^(1+2) * (-7) = -(-7) = 7. Wait, no, the cofactor for a12 is (-1)^(1+2) * (-7) = -(-7) = 7. Wait, no, the cofactor for a12 is (-1)^(1+2) * (-7) = -(-7) = 7. Wait, no, the cofactor for a12 is (-1)^(1+2) * (-7) = -(-7) = 7. Wait, no, the cofactor for a12 is (-1)^(1+2) * (-7) = -(-7) = 7. Wait, no, the cofactor for a12 is (-1)^(1+2) * (-7) = -(-7) = 7. Wait, no, the
GRPO success example (task_index 110, pass 3).	
user	Step 1: find the cofactor matrix for A = [[100, 99], [47, -7]]. Step 2: calculate the transpose of matrix B = the result from step 1. Step 3: what is the trace of C = the result from step 2?
assistant	<think>Okay, let's tackle this problem step by step. First, I need to find the cofactor matrix of matrix A. The matrix given is [[100, 99], [47, -7]]. The cofactor matrix involves calculating the minors and applying the appropriate signs based on the position. For a 2x2 matrix, the cofactor matrix can be found by swapping the diagonal elements and changing the signs of the off-diagonal elements. So, the cofactor of element a11 (100) is (-7), a12 (99) becomes -47, a21 (47) becomes -99, and a22 (-7) remains 100. But wait, I should use the matrix_cofactor tool to get the accurate result. Next, once I have the cofactor matrix, I need to transpose it. The transpose of a matrix swaps its rows and columns. So if the cofactor matrix is [[-7, -47], [-99, 100]], the transpose would be [[-7, -99], [-47, 100]]. Again, I'll use the matrix_transpose tool to confirm this. Finally, the trace of a matrix is the sum of its diagonal elements. For the transposed matrix, the diagonal elements are -7 and 100. Adding them gives 93. I'll use the matrix_trace tool to compute this. I need to make sure each step is done in order, using the correct tools. Let's start with the cofactor matrix.</think>

tool call	name=matrix_cofactor
	arguments="{\"matrix\": [[100, 99], [47, -7]]}"

tool	[[-7, -47], [-99, 100]]
assistant	<think>Okay, so the first step was to find the cofactor matrix of A, which the tool gave me as [[-7, -47], [-99, 100]]. Now, step 2 is to calculate the transpose of this matrix. The transpose of a matrix swaps its rows and columns. So, the element at position (1,2) becomes (2,1) and vice versa. Let me apply the matrix_transpose tool to this result. The original cofactor matrix is: Row 1: -7, -47 Row 2: -99, 100 Transposing it should give: Column 1: -7, -99 Column 2: -47, 100 So the transposed matrix should be [[-7, -99], [-47, 100]]. I'll use the matrix_transpose tool to confirm this. Once I have the transposed matrix, step 3 is to find its trace, which is the sum of the diagonal elements. The diagonal elements here are -7 and 100, so their sum is 93. But I need to use the matrix_trace tool to get the exact value. Let's proceed with the transpose first.</think>

tool call	name=matrix_transpose
	arguments="{\"matrix\": [[-7, -47], [-99, 100]]}"

tool	[[-7, -99], [-47, 100]]

Role	Message content (verbatim)
assistant	<think>Okay, let's see. The user has three steps to solve. First, find the cofactor matrix of A, which they did. Then, transpose that result, and finally, find the trace of the transposed matrix. So, the cofactor matrix was [[-7, -47], [-99, 100]]. Transposing that gives [[-7, -99], [-47, 100]]. Now, the trace is the sum of the diagonal elements. The diagonal elements here are -7 and 100. Adding them together should give 93. But I need to use the matrix_trace tool to confirm this. Let me call that tool with the transposed matrix as input.</think>

tool call	name=matrix_trace
	arguments="{\"matrix\": [[-7, -99], [-47, 100]]}"

tool	93.0
assistant	<think>Okay, let me go through the steps again to make sure everything is correct. The user asked for three steps: cofactor matrix, transpose, and trace. First, the cofactor matrix of A = [[100, 99], [47, -7]] was calculated. The cofactor matrix for a 2x2 matrix [[a, b], [c, d]] is [[d, -c], [-b, a]]. Applying that here, the cofactor matrix should be [[-7, -47], [-99, 100]], which matches the tool's response. That's correct. Next, transposing the cofactor matrix. The transpose of [[-7, -47], [-99, 100]] is [[-7, -99], [-47, 100]], which the tool also returned. That's right because transposing swaps rows and columns. Finally, the trace of the transposed matrix. The trace is the sum of the diagonal elements. The diagonal elements here are -7 and 100, so 93. The tool correctly returned 93.0. All steps check out. The final answer should be 93.0.</think> <answer>93.0</answer>
Answer tag missing (SFT eval, task_index 226, pass 1).	
Failure reason: Answer tag missing (final: missing <answer> tag).	
user	Find tr(A) for A = [[35, 10, 7], [-47, -26, 40], [-22, -7, -43]].
assistant	<think>Okay, I need to find the trace of matrix A. The trace is the sum of the diagonal elements. The matrix given is a 3x3 matrix. The diagonal elements are 35, -26, and -43. But instead of calculating it manually, I should use the matrix_trace tool. Let me call that tool with the provided matrix. The tool will handle the calculation, so I don't need to do any arithmetic myself. The result should be the sum of these three numbers.</think>

tool call	name=matrix_trace
	arguments="{\"matrix\": [[35, 10, 7], [-47, -26, 40], [-22, -7, -43]]}"

tool	-34.0
assistant	<think>Okay, so I need to find the trace of matrix A. The trace is the sum of the diagonal elements. The matrix given is a 3x3 matrix. The diagonal elements here are 35, -26, and -43. But instead of calculating it manually, I should use the matrix_trace tool. Let me call that tool with the provided matrix. The tool will handle the calculation, so I don't need to do any arithmetic myself. The result should be the sum of these three numbers.</think> <|im_start|> {"name": "matrix_trace", "arguments": "{\"matrix\": [[35, 10, 7], [-47, -26, 40], [-22, -7, -43]]}"} </tool_call>
GRPO success example (task_index 226, pass 1).	
user	Find tr(A) for A = [[35, 10, 7], [-47, -26, 40], [-22, -7, -43]].
assistant	<think>Okay, I need to find the trace of matrix A. The trace is the sum of the diagonal elements. The matrix given is a 3x3 matrix. The diagonal elements are 35, -26, and -43. But instead of calculating it manually, I should use the matrix_trace tool. Let me call that tool with the provided matrix. The tool will handle the calculation, so I just need to pass the matrix correctly.</think>

tool call	name=matrix_trace
	arguments="{\"matrix\": [[35, 10, 7], [-47, -26, 40], [-22, -7, -43]]}"

tool	-34.0
assistant	<think>Okay, so I need to find the trace of matrix A. The trace is the sum of the diagonal elements. The matrix given is a 3x3 matrix. The diagonal elements here are 35, -26, and -43. But instead of calculating it manually, I should use the matrix_trace tool. Let me call that tool with the provided matrix. The tool will handle the calculation, so I just need to pass the matrix correctly. The tool response should give the sum of these elements. Let me make sure the matrix is correctly formatted. The matrix is [[35, 10, 7], [-47, -26, 40], [-22, -7, -43]]. The diagonal elements are 35, -26, and -43. Adding them up: 35 + (-26) + (-43) = 35 -26 -43. Let me check the calculation. 35 minus 26 is 9, and 9 minus 43 is -34. So the trace should be -34.0. The tool's response is -34.0, which matches my calculation. So the answer is -34.0.</think> <answer>-34.0</answer>

Role	Message content (verbatim)
Answer unparseable (SFT eval, task_index 298, pass 2).	
Failure reason: Answer unparseable (Final: <answer> not parseable as a value).	
user	Find A^T for A = [[234, -603, -168], [-44, -47, -57], [-302, 474, 445]].
assistant	<think>Okay, I need to find the transpose of matrix A. The user provided the matrix A as a 3x3 matrix. The transpose of a matrix is obtained by swapping its rows with columns. So, the element in the first row and second column becomes the element in the second row and first column, and so on. Looking at the tools available, there's a matrix_transpose function that takes a matrix and returns its transpose. That's exactly what I need here. I don't need to calculate anything manually; I can just call the matrix_transpose tool with the given matrix as the argument. The tool should handle the swapping of rows and columns correctly. Let me make sure the matrix is correctly formatted as a list of rows. The input matrix is [[234, -603, -168], [-44, -47, -57], [-302, 474, 445]], which is a valid 3x3 matrix. So the correct function call is matrix_transpose with the provided matrix. I'll proceed to use the tool to get the transpose.</think>

tool call	name=matrix_transpose
	arguments="{\"matrix\": [[234, -603, -168], [-44, -47, -57], [-302, 474, 445]]}"

tool	[[234, -44, -302], [-603, -47, 474], [-168, -57, 445]]
assistant	<think>Okay, so the user asked for the transpose of matrix A. I used the matrix_transpose tool with the given matrix. The tool returned the transposed matrix. Let me check if that makes sense. The original matrix A is: Row 1: [234, -603, -168] Row 2: [-44, -47, -57] Row 3: [-302, 474, 445] Transposing it means swapping rows and columns. So the first column of the original becomes the first row of the transpose. Let's verify: Original first column: 234, -44, -302. Transposed becomes the first row: [234, -44, -302]. Original second column: -603, -47, 474. Transposed becomes the second row: [-603, -47, 474]. Original third column: -168, -57, 445. Transposed becomes the third row: [-168, -57, 445]. Wait, the tool's response is [[234, -44, -302], [-603, -47, 474], [-168, -57, 445]]. Hmm, that looks correct. The first row is the original first column, the second row is the original second column, and the third row is the original third column. So the tool's output is correct. No mistakes here. The user probably just wanted the transpose, so the answer is the transposed matrix as provided by the tool.</think> <answer>234, -44, -302 -603, -47, 474 -168, -57, 445</answer>
GRPO success example (task_index 298, pass 3).	
user	Find A^T for A = [[234, -603, -168], [-44, -47, -57], [-302, 474, 445]].
assistant	<think>Okay, I need to find the transpose of matrix A. The user provided the matrix A as a 3x3 matrix. The transpose of a matrix is obtained by swapping its rows with columns. So, the element in the first row and second column becomes the element in the second row and first column, and so on for all elements. Looking at the tools available, there's a matrix_transpose function that takes a matrix and returns its transpose. That's exactly what I need here. I don't need to calculate anything manually; I can just call the matrix_transpose tool with the given matrix as the argument. The tool should handle the swapping of rows and columns correctly. Let me make sure the matrix is correctly formatted as a list of rows. The input matrix is [[234, -603, -168], [-44, -47, -57], [-302, 474, 445]], which is a valid 3x3 matrix. So the correct function call is matrix_transpose with the provided matrix. I'll proceed to use the tool now.</think>

tool call	name=matrix_transpose
	arguments="{\"matrix\": [[234, -603, -168], [-44, -47, -57], [-302, 474, 445]]}"

tool	[[234, -44, -302], [-603, -47, 474], [-168, -57, 445]]

Role	**Message content (verbatim)**
assistant	<think>Okay, so the user asked for the transpose of matrix A. I used the matrix_transpose tool with the given matrix. The tool returned the transposed matrix. Let me check if that makes sense. The original matrix A is: Row 1: [234, -603, -168] Row 2: [-44, -47, -57] Row 3: [-302, 474, 445] Transposing it means swapping rows and columns. So the first column of the original becomes the first row of the transpose. Let me verify: Original first column: 234, -44, -302. Transposed becomes the first row: [234, -44, -302]. Original second column: -603, -47, 474. Transposed becomes the second row: [-603, -47, 474]. Original third column: -168, -57, 445. Transposed becomes the third row: [-168, -57, 445]. Wait, the tool's response is [[234, -44, -302], [-603, -47, 474], [-168, -57, 445]]. Hmm, the second row is correct, but the third column in the original is -168, -57, 445. The tool's third row is [-168, -57, 445], which matches. So the transposed matrix looks correct. No errors in the tool's output. The user probably just wanted the transpose, so the answer should be the transposed matrix as provided by the tool.</think> <answer>[[234, -44, -302], [-603, -47, 474], [-168, -57, 445]]</answer>
Format bad (SFT eval, task_index 387, pass 3).	
Failure reason: Format bad (Malformed <tool_call>/<answer> tags).	
user	Calculate the transpose of matrix A = [[184, -592], [399, -374]].
assistant	<think>Okay, I need to find the transpose of matrix A. The matrix given is [[184, -592], [399, -374]]. Transposing a matrix means swapping its rows and columns. So, the element in the first row, second column becomes the element in the second row, first column, and vice versa. Looking at the tools available, there's a matrix_transpose function. The parameters require the matrix as a list of rows. The input matrix here is 2x2, so it should work. I just need to pass the matrix to the matrix_transpose tool. The result should be the transposed matrix. Let me call the tool with the given matrix.</think>

tool call	name=matrix_transpose
	arguments="{\"matrix\": [[184, -592], [399, -374]]}"

tool	[[184, 399], [-592, -374]]
assistant	<think>Okay, so I need to find the transpose of matrix A, which is [[184, -592], [399, -374]]. The transpose of a matrix is obtained by swapping its rows and columns. Let me verify that. The original matrix has rows [184, -592] and [399, -374]. The first row becomes the first column in the transpose, and the second row becomes the second column. So the transposed matrix should have the first column as [184, 399] and the second column as [-592, -374]. Looking at the tool's response, it returns [[184, 399], [-592, -374]]. That matches my expectation. The tool's output is correct. Therefore, the answer is the transposed matrix as provided by the tool.</think> <answer> {"name": "matrix_transpose", "arguments": "{\"matrix\": [[184, -592], [399, -374]]}"} </tool_call>
GRPO success example (task_index 387, pass 2).	
user	Calculate the transpose of matrix A = [[184, -592], [399, -374]].
assistant	<think>Okay, I need to find the transpose of matrix A. The matrix given is [[184, -592], [399, -374]]. Transposing a matrix means swapping its rows and columns. So the element in the first row, second column becomes the element in the second row, first column, and vice versa. The matrix_transpose tool should handle this. Let me call that tool with the given matrix.</think>

tool call	name=matrix_transpose
	arguments="{\"matrix\": [[184, -592], [399, -374]]}"

tool	[[184, 399], [-592, -374]]
assistant	<think>Okay, so the user asked for the transpose of matrix A. I used the matrix_transpose tool with the given matrix. The tool returned [[184, 399], [-592, -374]]. Let me check if that's correct. The original matrix has rows [184, -592] and [399, -374]. Transposing should swap these rows and columns. So the first column of the original becomes the first row of the transpose: 184 and 399. The second column becomes the second row: -592 and -374. That matches the tool's output. So the answer is correct. No further steps needed.</think> <answer>[[184, 399], [-592, -374]]</answer>

Role	**Message content (verbatim)**
Turn deviation (SFT eval, task_index 271, pass 3).	
Failure reason: Turn deviation (Tool-call count: expected N, got $N \pm 1$).	
user	Find tr(A) for A = [[-23, -3], [-12, 25]].
assistant	<think>Okay, I need to find the trace of matrix A. The trace is the sum of the diagonal elements. The matrix given is [[-23, -3], [-12, 25]]. The diagonal elements here are -23 and 25. So I should add those two numbers together. But wait, the user wants me to use the matrix_trace tool. Let me check the parameters. The tool requires the matrix as a list of rows. The input matrix is already in that format. So I can directly call the matrix_trace function with the given matrix. The result should be -23 + 25, which is 2. But I need to make sure I use the tool correctly. The tool will handle the calculation, so I just need to pass the matrix correctly. Let me structure the tool call properly.</think>

tool call	name=matrix_trace
	arguments="{\"matrix\": [[-23, -3], [-12, 25]]}"

tool	2.0
assistant	<think>Okay, so I need to find the trace of matrix A, which is given as [[-23, -3], [-12, 25]]. The trace is the sum of the diagonal elements. The diagonal elements here are -23 and 25. Adding them together, -23 + 25 equals 2. But I should use the matrix_trace tool to confirm this. Let me call the tool with the matrix provided. The tool will handle the calculation, so I just need to pass the matrix correctly. The result should be 2.0. I'll make sure to use the correct parameters and structure the tool call properly.</think>

tool call	name=matrix_trace
	arguments="{\"matrix\": [[-23, -3], [-12, 25]]}"

tool	2.0
assistant	<think>Okay, let me see. The user asked for the trace of matrix A, which is [[-23, -3], [-12, 25]]. The trace is the sum of the diagonal elements. The diagonal elements here are -23 and 25. Adding them together, -23 + 25 equals 2. The matrix_trace tool was called with the correct matrix, and it returned 2.0. That makes sense. So the answer should be 2.0. I need to make sure there's no other step required, but since the trace is directly calculated, the final answer is 2.0.</think> <answer>2.0</answer>
GRPO success example (task_index 271, pass 2).	
user	Find tr(A) for A = [[-23, -3], [-12, 25]].
assistant	<think>Okay, I need to find the trace of matrix A. The trace is the sum of the diagonal elements. The matrix given is [[-23, -3], [-12, 25]]. The diagonal elements here are -23 and 25. So I should add those two numbers together. But wait, the user wants me to use the matrix_trace tool. Let me check the parameters. The tool requires the matrix as a list of rows. The input matrix is already in that format. So I can directly call the matrix_trace function with the given matrix. The result should be -23 + 25, which is 2. But I need to make sure I use the tool correctly. The tool will handle the calculation, so I just need to pass the matrix correctly. Let me structure the tool call properly.</think>

tool call	name=matrix_trace
	arguments="{\"matrix\": [[-23, -3], [-12, 25]]}"

tool	2.0
assistant	<think>Okay, so the user asked for the trace of matrix A, which is [[-23, -3], [-12, 25]]. The trace is the sum of the diagonal elements. The diagonal elements here are -23 and 25. Adding them together, -23 + 25 equals 2. The matrix_trace tool was used, and it returned 2.0. That matches my calculation. Since the tool's result is correct, the answer should be 2.0. No further steps needed.</think> <answer>2.0</answer>

Table C.1: Representative SFT failure traces paired with GRPO successes for each exclusive error category in Table 6.1. System prompts are omitted for brevity.