# AI Agents for Science

Lecture 13, November 10

Building Agents and Workflows

Instructor: Ian Foster

TA: Alok Kamatar

*Crescat scientia; vita excolatur*

CMSC 35370 -- https://agents4science.github.io

https://canvas.uchicago.edu/courses/67079

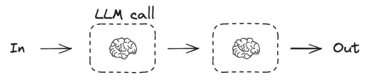
# Recall: Workflows vs. agents

https://langchain-ai.github.io/langgraph/tutorials/workflows/
https://www.anthropic.com/engineering/building-effective-agents

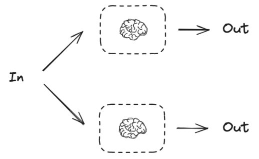
**Workflows:** Systems where LLMs and tools are orchestrated through predefined code paths.
**Agents: S**ystems where LLMs dynamically direct their own processes and tool usage, maintaining control over how they accomplish tasks.

## Workflows

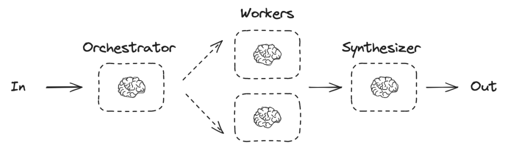
### Prompt Chaining



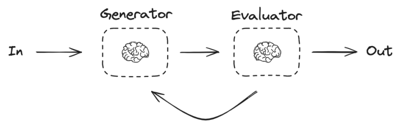
### Parallelization



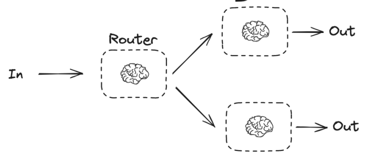
### Orchestrator-Worker



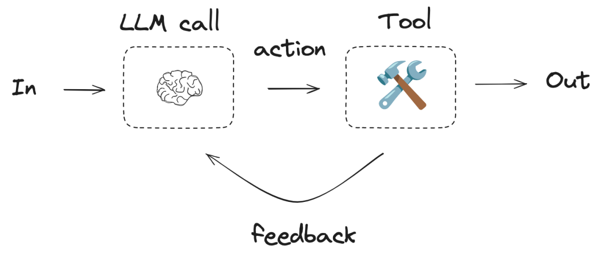
### Evaluator-optimizer



### Routing



## Agent



LLM is embedded in predefined code paths

LLM directs control flow through predefined code paths

LLM directs its own actions based on environmental feedbac

## Building agents and workflows: Readings

[AFlow: Automating Agentic Workflow Generation]

[DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines]

## A simple example of an “AI workflow”

**Task:** Summarize a research paper, then generate three research questions based on the summary, and rank the questions

1) Summarization

   **Prompt**: “Summarize the following paper in one paragraph: {paper_text}”

   **Output**: “This paper introduces a transformer model for protein folding …”

2) Question generation

   **Prompt**: “Based on this summary, generate three research questions: {summary_from_step1}”

   **Output**: “How can we improve folding accuracy?”, etc.

3) Ranking

   **Prompt**: “Rank these research questions by novelty and feasibility: {questions_from_step2}”

   **Output**: “Q2 > Q1 > Q3.”

## Critique

- **Fixed order and format:** If step 1 produces too long or too short a summary, step 2 fails or generates incoherent questions
- **No feedback or self-correction:** The system never checks whether the questions actually relate to the paper
- **No adaptation:** It cannot adjust its strategy if, e.g., the paper is mathematical, biological, or philosophical
- **Manual tuning:** Each prompt must be hand-crafted, and a small change in model behavior breaks the pipeline

## Building robust agents and workflows

- Recent systems aim to **automate the composition, optimization, and adaptation** of pipelines
- Goal: turn “hand-crafted orchestration” into **self-configuring, learning workflows** that continuously improve with use
- Key ideas:
  - Represent agentic workflows declaratively rather than imperatively
  - Have workflows introspect on their own performance and improve automatically
  - Apply data, evaluation, and compiler-like abstractions to self-improvement

## Example of a robust pipeline

**Goal:** Summarize a paper, generate research questions, and return only high-quality, on-topic, non-redundant questions

- Ingest & sanity checks
- Grounded summarization (with retrieval)
- Question generation (diversity-controlled)
- Scoring & filtering
- Refinement loop (bounded)
- Final selection & packaging

- Detect format (PDF/HTML), extract text, deduplicate pages, and chunk long inputs
- Validate language and length; if too short/long, adjust chunking strategy

# Example of a robust pipeline

**Goal:** Summarize a paper, generate research questions, and return only high-quality, on-topic, non-redundant questions

- Ingest & sanity checks
- Grounded summarization (with retrieval)
- Question generation (diversity-controlled)
- Scoring & filtering
- Refinement loop (bounded)
- Final selection & packaging

- Build a lightweight index over the paper chunks
- Prompt model to produce a sectioned summary (Background/Method/Results/Limitations) with **citations to chunk IDs**.
- **Guardrail:** A rule-based/LLM verifier checks that each claim is attributable to cited chunks; if not, re-prompt with stricter instructions or reduce temperature

## Example of a robust pipeline

**Goal:** Summarize a paper, generate research questions, and return only high-quality, on-topic, non-redundant questions

- Ingest & sanity checks
- Grounded summarization (with retrieval)
- Question generation (diversity-controlled)
- Scoring & filtering
- Refinement loop (bounded)
- Final selection & packaging

- Ask for K candidate questions across **categories** (clarification, extension, evaluation)
- Enforce a schema (JSON with category, question, evidence_chunks)
- **Guardrail:** Deduplicate by embedding similarity; drop near-duplicates

# Example of a robust pipeline

**Goal:** Summarize a paper, generate research questions, and return only high-quality, on-topic, non-redundant questions

- Ingest & sanity checks
- Grounded summarization (with retrieval)
- Question generation (diversity-controlled)
- Scoring & filtering
- Refinement loop (bounded)
- Final selection & packaging

- Compute scores for **relevance**, **novelty**, and **feasibility** using small scorers (could be LLM-as-judge or lightweight heuristics)
- Apply thresholds; if average score < τ, **retry**: adjust prompt (e.g., emphasize methods/results) or **route** to a stronger model for a second pass

## Example of a robust pipeline

**Goal:** Summarize a paper, generate research questions, and return only high-quality, on-topic, non-redundant questions

- Ingest & sanity checks
- Grounded summarization (with retrieval)
- Question generation (diversity-controlled)
- Scoring & filtering
- Refinement loop (bounded)
- Final selection & packaging

- For any low-scoring question, request a **rewrite** citing specific paper sections
- Stop after N iterations or upon meeting quality thresholds

## Example of a robust pipeline

**Goal:** Summarize a paper, generate research questions, and return only high-quality, on-topic, non-redundant questions

- Ingest & sanity checks
- Grounded summarization (with retrieval)
- Question generation (diversity-controlled)
- Scoring & filtering
- Refinement loop (bounded)
- Final selection & packaging

- Return top 3–5 questions with short rationales and the chunk citations that support them
- Log prompt variants, scores, and decisions for **self-improvement** later (e.g., tune prompts or module order based on outcomes)

## Example of a robust pipeline

**Goal:** Summarize a paper, generate research questions, and return only high-quality, on-topic, non-redundant questions

- Ingest & sanity checks
- Grounded summarization (with retrieval)
- Question generation (diversity-controlled)
- Scoring & filtering
- Refinement loop (bounded)
- Final selection & packaging

**Resilience features baked in**

- **Schema validation** (rejects malformed outputs)
- **Attribution checks** (prevents hallucinations)
- **Retries with strategy shifts** (temperature, prompt, or model)
- **Routing/fallback** (use a cheaper model first, escalate only if needed)
- **Bounded loops & timeouts** (no infinite retries)
- **Caching** (reuse summaries/scores when re-run on similar inputs)
- **Telemetry** (store metrics to learn better defaults over time)

## DSPy (= Demonstrate-Search-Predict)

- DSPy is an open-source Python framework that allows developers to build language model applications using modular and declarative programming instead of relying on one-off prompting techniques
- Instead of free-form string prompts, DSPy programs use **natural language signatures** to assign work to the LM. A DSPy signature is natural-language typed declaration of a function: a short declarative spec that tells DSPy what a text transformation needs to do (e.g., “consume questions and return answers”), rather than how a specific LM should be prompted to implement that behavior. More formally, a DSPy signature is a tuple of input fields and output fields (and an optional instruction).

https://www.datacamp.com/blog/dspy-introduction

## Simple question-answer pattern

```python
qa = dspy.Predict("question -> answer")
qa(question="Where is Guaraní spoken?")
# Out: Prediction(answer='Guaraní is spoken mainly in South America.')
```

## Chain of thought

```python
class ChainOfThought(dspy.Module):
    def __init__(self, signature):
        # Modify signature from ‘*inputs -> *outputs‘ to ‘*inputs -> rationale, *outputs‘.
        rationale_field = dspy.OutputField(prefix="Reasoning: Let's think step by step.")
        signature = dspy.Signature(signature).prepend_output_field(rationale_field)

        # Declare a sub-module with the modified signature.
        self.predict = dspy.Predict(signature)

    def forward(self, **kwargs):
        # Just forward the inputs to the sub-module.
        return self.predict(**kwargs)
```

## Compilation and optimizers

A DSPy program thus provides a signature for a prompt

“Compiling” a DSPy program means running an optimizer (a *teleprompter*) that takes **your program + a training set + a metric** and searches for better prompts/parameters

That search often includes **choosing (or synthesizing) few-shot examples** to place in the prompt for each module

The optimizer simulates your pipeline on the training data, evaluates candidates, and keeps the instruction text and example set that score best

## Retrieval augmented generation

```python
class RAG(dspy.Module):
    def __init__(self, num_passages=3):
        # ‘Retrieve‘ will use the user’s default retrieval settings unless overriden.
        self.retrieve = dspy.Retrieve(k=num_passages)
        # ‘ChainOfThought‘ with signature that generates answers given retrieval & question.
        self.generate_answer = dspy.ChainOfThought("context, question -> answer")

    def forward(self, question):
        context = self.retrieve(question).passages
        return self.generate_answer(context=context, question=question)
```

## Bootstrapping

The following code compiles the RAG module against a dataset of question-answer pairs, qa_trainset, to bootstrap few-shot demonstrations:

```python
# Small training set with only questions and final answers.
qa_trainset = [dspy.Example(question="What is the capital of France?", answer="Paris")]

# The teleprompter will bootstrap missing labels: reasoning chains and retrieval contexts.
teleprompter = dspy.BootstrapFewShot(metric=dspy.evaluate.answer_exact_match)
compiled_rag = teleprompter.compile(RAG(), trainset=qa_trainset)
```

**Compiling** As we discussed in Section 4, DSPy programs can be compiled into new, optimized programs. In our experiments, we evaluate the programs zero-shot (no compiling) as well as a number of strategies for compiling. Our simplest compiler is `LabeledFewShot`:

```
fewshot = dspy.LabeledFewShot(k=8).compile(program, trainset=trainset)
```

Here, `program` can be any DSPy module. This simply samples `k=8` random demonstrations from the `trainset` for the fields common to the training examples and the signature(s), in this case, `question` and `answer`, but not the reasoning for instance. We report the average of 3–5 runs (depending on the setting) when applying such random sampling.

Teleprompters can be composed by specifying a `teacher` program. DSPy will sample demonstrations from this program for prompt optimization. This composition can enable very rich pipelines, where expensive programs (e.g., complex expensive ensembles using large LMs) supervise cheap programs (e.g., simple pipelines using smaller LMs). One may start with `compiled_rag` from above (say, compiled to use a large Llama2-13b-chat LM) but now fine-tune Flan-T5-large to create an efficient program:

```
# Larger set of questions with *no labels*. Labels for all steps will be bootstrapped.
unlabeled_questions = [dspy.Example(question="What is the capital of Germany?"), ...]

# As we assumes no answer, we use ‘answer_passage_match‘ to filter ungrounded answers.
finetuning_teleprompter = BootstrapFinetune(metric=dspy.evaluate.answer_passage_match)

# We set ‘teacher=compiled_rag‘ to compose. Bootstrapping will now use ‘compiled_rag‘.
compiled_rag_via_finetune = finetuning_teleprompter.compile(RAG(), teacher=compiled_rag,
    trainset=unlabeled_questions, target='google/flan-t5-large')
```

## BootstrapFewShot

```python
class SimplifiedBootstrapFewShot(Teleprompter):
    def __init__(self, metric=None):
        self.metric = metric

    def compile(self, student, trainset, teacher=None):
        teacher = teacher if teacher is not None else student
        compiled_program = student.deepcopy()

        # Step 1. Prepare mappings between student and teacher Predict modules.
        # Note: other modules will rely on Predict internally.
        assert student_and_teacher_have_compatible_predict_modules(student, teacher)
        name2predictor, predictor2name = map_predictors_recursively(student, teacher)

        # Step 2. Bootstrap traces for each Predict module.
        # We'll loop over the training set. We'll try each example once for simplicity.
        for example in trainset:
            if we_found_enough_bootstrapped_demos(): break

            # turn on compiling mode which will allow us to keep track of the traces
            with dspy.setting.context(compiling=True):
                # run the teacher program on the example, and get its final prediction
                # note that compiling=True may affect the internal behavior here
                prediction = teacher(**example.inputs())

                # get the trace of the all interal Predict calls from teacher program
                predicted_traces = dspy.settings.trace

            # if the prediction is valid, add the example to the traces
            if self.metric(example, prediction, predicted_traces):
                for predictor, inputs, outputs in predicted_traces:
                    d = dspy.Example(automated=True, **inputs, **outputs)
                    predictor_name = self.predictor2name[id(predictor)]
                    compiled_program[predictor_name].demonstrations.append(d)


        return compiled_program
```

## BootstrapFewShotWithRandomSearch

```python
class SimplifiedBootstrapFewShotWithRandomSearch(Teleprompter):
    def __init__(self, metric = None, trials=16):
        self.metric = metric
        self.trials = trials

    def compile(self, student, *, teacher=None, trainset, valset=None):
        # we can do forms of cross-validation if valset is unset.
        valset = trainset if valset is None else valset

        candidates = []
        for seed in range(self.trials):
            # Create a new basic bootstrap few-shot program.
            shuffled_trainset = shuffle(trainset, seed=seed)
            tp = BootstrapFewShot(metric=metric, max_bootstrap_demos=random_size())
            candidate_program = tp.compile(student, shuffled_trainset, teacher)

            # Step 2: Evaluate the generated candidate program.
            score = evaluate_program(candidate_program, self.metric, valset)
            candidates.append((score, candidate_program))

        # return the best candidate program.
        return max(candidates, key=lambda x: x[0])[1]
```

## Evaluation

Goal is to explore the role of hand-written, task-specific prompts in achieving performant systems. Seek to test three hypotheses:

**H1:** With DSPy, we can replace hand-crafted prompt strings with concise and well-defined modules, without reducing quality or expressive power

**H2:** Parameterizing the modules and treating prompting as an optimization problem makes DSPy better at adapting to different LMs, and it may outperform expert-written prompts

**H3:** The resulting modularity makes it possible to more thoroughly explore complex pipelines that have useful performance characteristics or that fit nuanced metrics

## A grand goal

- We hope this begins a shift from underspecified questions like “how do different LMs compare on GSM8K” toward “how they compare on GSM8K with program P when compiled with strategy S”, which is a well-defined and reproducible run

- Ultimately, our goal is to reduce the role of artful prompt construction in modern AI in favor of the development of new modular, composable programs and optimizers

We evaluate on the popular GSM8K dataset with grade school math questions (Cobbe et al., 2021). We sample 200 and 300 question–answer pairs from the official training set for training and development, respectively. Our final evaluations use the 1.3k official test set examples. We report extensive comparisons on the development set to avoid overfitting on test. Following prior work on GSM8K, we evaluate the accuracy of the final numerical value that appears in the LM output.

**Programs Considered** For this task, we consider three simple DSPy programs: a one-step Predict module (`vanilla`), a two-step ChainOfThought module (`CoT`), and finally a multi-stage ComparerOfThoughts module (`ThoughtReflection`). These are fully defined by the following code:

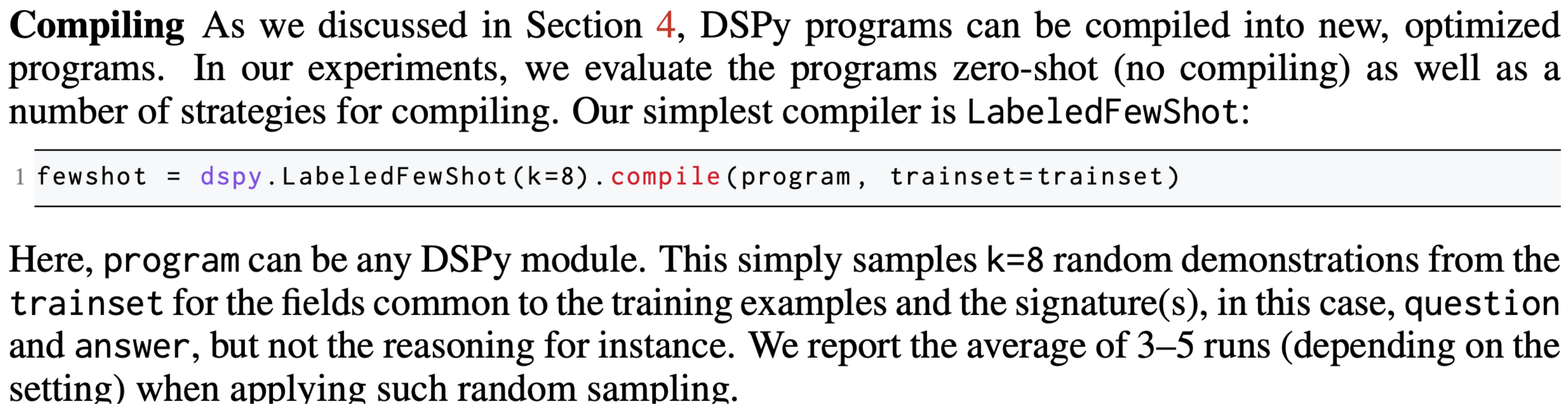
```python
vanilla = dspy.Predict("question -> answer")  # GSM8K Program 'vanilla'

CoT = dspy.ChainOfThought("question -> answer")  # GSM8K Program 'CoT'
```

```python
class ThoughtReflection(dspy.Module):
    def __init__(self, num_attempts):
        self.predict = dspy.ChainOfThought("question -> answer", n=num_attempts)
        self.compare = dspy.MultiChainComparison('question -> answer', M=num_attempts)

    def forward(self, question):
        completions = self.predict(question=question).completions
        return self.compare(question=question, completions=completions)

reflection = ThoughtReflection(num_attempts=5) # GSM8K Program 'reflection'
```

Samples 5 reasoning chains from the LM (+ answers) and compares in parallel

"Compile" means running an optimizer (a *teleprompter*) that takes **your program + a training set + a metric** and searches for better prompts/parameters. That search often includes **choosing (or synthesizing) few-shot examples** to place in the prompt for each module. In practice, the optimizer simulates your pipeline on the training data, evaluates candidates, and keeps the instruction text and example set that score best.

DSPy optimizers can **tune prompts and/or LM weights** and many explicitly **generate or select demonstrations** (e.g., *BootstrapFewShot*, *LabeledFewShot*, *MIPROv2*) during compile. The optimizer typically **runs the program on the training split**, scores outputs with your metric (e.g., EM/F1), and updates instructions + examples accordingly.



**Compiling** As we discussed in Section 4, DSPy programs can be compiled into new, optimized programs. In our experiments, we evaluate the programs zero-shot (no compiling) as well as a number of strategies for compiling. Our simplest compiler is `LabeledFewShot`:

```
fewshot = dspy.LabeledFewShot(k=8).compile(program, trainset=trainset)
```

Here, `program` can be any DSPy module. This simply samples `k=8` random demonstrations from the `trainset` for the fields common to the training examples and the signature(s), in this case, `question` and `answer`, but not the reasoning for instance. We report the average of 3–5 runs (depending on the setting) when applying such random sampling.

Table 1: Results with in-context learning on GSM8K math word problems. Each row represents a separate pipeline: the module in the Program column is compiled against the examples in the Training set. The programs, compilers, and (small) training sets are defined in Section 6. Rows with `ensemble` build on the immediately preceding row. Notably, all programs in this table are expressed by composing two to four DSPy modules and teleprompters. Compiling the correct *modules*, instead of string prompts, improves different LMs from 4–20% accuracy to 49–88% accuracy.

| | | | GPT-3.5 | | Llama2-13b-chat | |
|---|---|---|---|---|---|---|
| Program | Compilation | Training | Dev | Test | Dev | Test |
| `vanilla` | none | n/a | 24.0 | 25.2 | 7.0 | 9.4 |
| | `fewshot` | `trainset` | 33.1 | – | 4.3 | – |
| | `bootstrap` | `trainset` | 44.0 | – | 28.0 | – |
| | `bootstrap`×2 | `trainset` | 64.7 | 61.7 | 37.3 | 36.5 |
| | `+ensemble` | `trainset` | 62.7 | 61.9 | 39.0 | 34.6 |
| `CoT` | none | n/a | 50.0 | – | 26.7 | – |
| | `fewshot` | `trainset` | 63.0 | – | 27.3 | – |
| | `fewshot` | `+human_CoT` | 78.6 | 72.4 | 34.3 | 33.7 |
| | `bootstrap` | `trainset` | 80.3 | 72.9 | 43.3 | – |
| | `+ensemble` | `trainset` | **88.3** | 81.6 | 43.7 | – |
| `reflection` | none | n/a | 65.0 | – | 36.7 | – |
| | `fewshot` | `trainset` | 71.7 | – | 36.3 | – |
| | `bootstrap` | `trainset` | 83.0 | 76.0 | 44.3 | 40.2 |
| | `+ensemble` | `trainset` | 86.7 | – | **49.0** | **46.9** |

## AFlow: Automating Agentic Workflow Generation

- AFlow treats the design of agentic workflows (i.e., sequences or graphs of LLM-invocation nodes + tool calls + reasoning steps) as a **search problem**
- It uses techniques like **Monte Carlo Tree Search (MCTS)** over a space of code-represented workflows: each workflow is represented as a graph of nodes (LLM or tool calls) plus edges (flow of data) and AFlow searches that space to find high-performing workflows.
- Empirical results: across QA, Code, Math datasets, workflows found by AFlow improved over manually-designed workflows (average +5.7% improvement) and allowed smaller models to outperform larger ones in cost-performance trade-off
- Use case: When your problem requires designing *the workflow structure itself* (not just prompt templates): e.g., deciding how many reasoning steps, the tool calls, intermediate verification steps, ensemble nodes, etc.

## Example: Scientific question answering

- You want an AI system to answer scientific questions such as:

  "Why does salt lower the freezing point of water?"

- You have:
  - Access to a retrieval API: search_papers(query)
  - A summarization LLM
  - A reasoning LLM
  - Optionally, a verifier LLM that checks factual consistency
- You don't know *which sequence* of these modules (or how many steps) yields the best factual accuracy

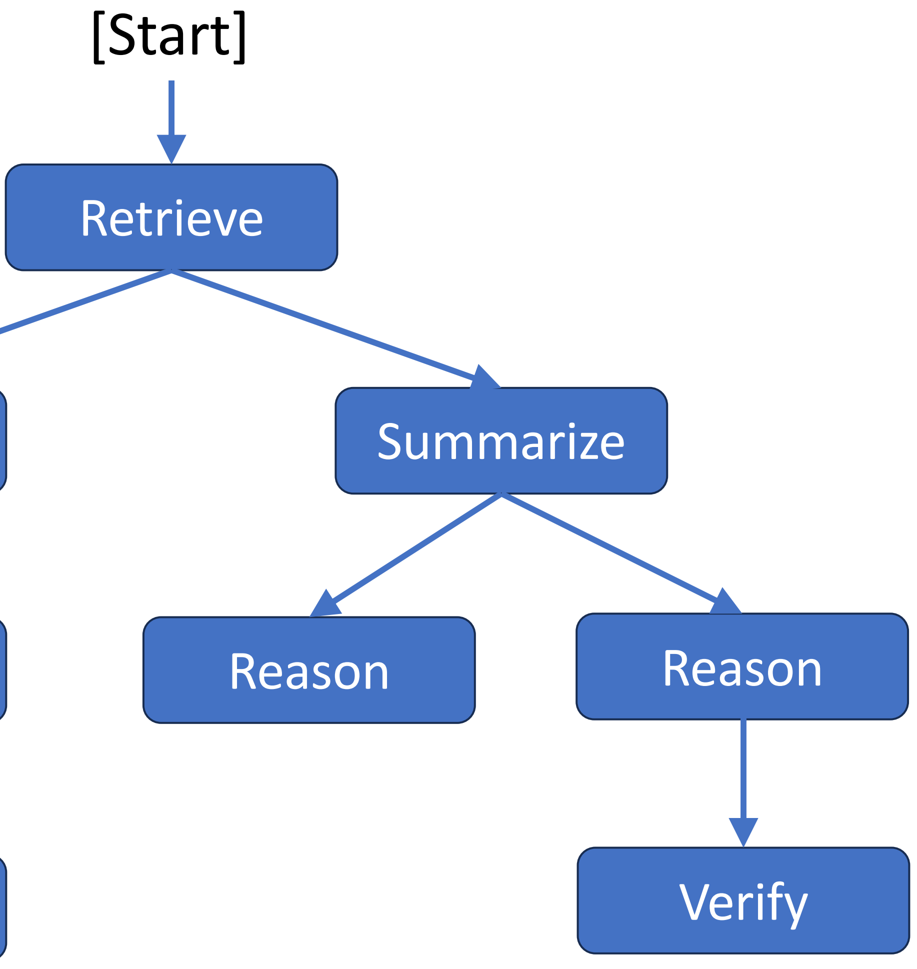
## Define a function library

```python
def retrieve(query):
    return search_papers(query, top_k=5)

def summarize(texts):
    return LLM("Summarize the key findings:", texts)

def reason(question, context):
    return LLM(f"Answer the question based on this context:\n{context}\nQ: {question}\nA:")

def verify(question, answer):
    return LLM(f"Is the following answer correct? Q:{question} A:{answer}")
```

## AFlow search strategy

- AFlow treats each **function** (LLM or tool) as a node, and possible **data flows** as edges
- It automatically constructs and explores candidate workflows like:
  - **Simple chain:** retrieve → summarize → reason
  - **Direct reasoning:** retrieve → reason
  - **Verification branch:** retrieve → summarize → reason → verify
  - **Two-pass reasoning:** retrieve → reason → reason
  - **Parallel ensemble:** ($\text{summarize}_1$ | $\text{summarize}_2$) → reason
- Each is represented internally as a graph. AFlow uses **Monte Carlo Tree Search (MCTS)** to explore combinations, guided by their performance on a validation set

## Monte Carlo Tree Search



## Automatic Evaluation

For each candidate workflow:

- AFlow **executes it** on a small training dataset (e.g., 100 science questions with gold answers)
- Compares each workflow's outputs using a scoring metric, e.g., F1 or factual accuracy from GPT-4 evaluation
- The **reward** (accuracy) guides the search policy, favoring sub-graphs that yield better answers
- Promising workflows are expanded or mutated (like AutoML for pipelines)

## Discovered workflow example

After several search rounds, AFlow might find that the **best-performing workflow** is:

$$\begin{tikzcd} \text{retrieve} \arrow[d] \\ \text{summarize} \arrow[d] \\ \text{reason} \arrow[d] \\ \text{verify} \arrow[dr, "\text{(if confidence} < 0.8\text{)}"] \\ & \text{reason\_again} \end{tikzcd}$$

This “verified double-pass” pipeline turns out to outperform human-crafted workflows: for example, achieving 78% factual accuracy versus 70% from baseline

## Export and reuse

Once found, the workflow can be **exported as runnable code** or a JSON graph:

```
{
  "nodes": ["retrieve", "summarize", "reason", "verify"],
  "edges": [
    ["retrieve", "summarize"],
    ["summarize", "reason"],
    ["reason", "verify"],
    ["verify", "reason"]
  ]
}
```

You can then plug this into your LLM orchestrator (LangGraph, AutoGen, etc.), or even wrap it as a DSPy program for further fine-tuning of prompts or weights

## Summary of actions taken

| **Stage** | **What Happened** |
|---|---|
| Define | You provided building blocks (functions / modules) |
| Search | AFlow explored possible graph structures of these modules |
| Evaluate | Each workflow was tested on real examples |
| Optimize | MCTS-based controller improved workflows by reward feedback |
| Output | The best-performing workflow (a verified reasoning chain) |

## AFlow and DSPy compared

| Concept | AFlow | DSPy | Common Thread |
|---|---|---|---|
| Representation | Agentic workflows, roles, plans | Declarative task specs, modules | Structured abstraction for composition |
| Improvement Mechanism | Self-refinement via meta-feedback | End-to-end optimization over evaluation metric | Self-improving systems |
| Analogy | AutoML for workflows | Compiler for LLM pipelines | Automation of pipeline design and tuning |
| Outcome | Adaptive, re-composable workflows | Optimized, reusable LLM components | “Self-engineering” systems |

## Future diections

- **Meta-agents for pipeline synthesis:** agents that invent new evaluation metrics, interfaces, and modular abstractions
- **Cross-domain reuse:** workflows that generalize across tasks via learned schemas
- **Integration with scientific computing:** applying these ideas to automate experiment planning, simulation–analysis loops, and lab workflows
- **Reflexive evaluation:** systems that improve their own *self-improvement* mechanisms (meta-optimization)

# AI Agents for Science

Lecture 14, November 12

Finetuning andreinforcement learning

Instructor: Ian Foster

TA: Alok Kamatar

*Crescat scientia; vita excolatur*

CMSC 35370 -- https://agents4science.github.io

https://canvas.uchicago.edu/courses/67079

## Reinforcement learning papers

Adapting agents with reinforcement learning and real-world training.

OpenPipe/ART: Agent Reinforcement Trainer

Agent Lightning: Train ANY AI Agents with Reinforcement Learning

## Recall: An agent is …

- An agent is a system that:
  - **Senses** (reads inputs, environment, or tool responses)
  - **Plans** (decides what to do next)
  - **Acts** (calls a tool)
  - **Learns** (updates state )
- Its operations are governed by policy that governs how it chooses *actions* in response to *states* (its current context)
- An LLM/RM may be used in various contents

## Possible roles of LLMs/RMs in agents

| **Stage** | **Role / Function** | **Possible LLM Involvement** |
|---|---|---|
| Sense | Observe the environment: read user inputs, tool outputs, or API responses | LLM interprets or summarizes current context (e.g., "What did the tool return?") |
| Plan | Decide *what to do next*: which subgoal or tool to use, how to proceed | LLM generates next-step plans or chain-of-thought reasoning |
| Act | Execute chosen step: call a function, run a tool, or generate a response | LLM issues structured commands or final answers |
| Learn | Improve based on results, rewards, or feedback | RL or prompt optimization adjusts LLM behavior or policy |

## Improving agent performance

- Goal: alter agent behavior, e.g., improves some aspect of performance
- Generally we want to do this by training, which may involve:
  - **LLM fine-tuning**: Presenting agent LLM with structured data (e.g., prompt/answer pairs) and adjusting LLM parameters to increase match to answer
  - **Reinforcement learning**: Running the agent in a real or simulated environment and adjusting aspects of the agent implementation (LLM, other policy components) to improve the reward obtained
- Training can allow the agent to:
  - Choose better next actions (more optimal tool calls, reasoning steps, responses)
  - Improve reliability and efficiency over repeated interactions
  - Learn from *experience*, not just static data

## Levels at which performance may be improved

| **Level** | **Object being improved** | **Typical method** | **What changes** |
|---|---|---|---|
| Model (LLM) | A single neural policy that maps *text* → *text* | Fine-tuning (supervised or RL-based) | Model parameters or adapters |
| Agent implement. | A *system* that uses one or more LLMs plus tools, memory, control logic | Training loop/RL algorithm (e.g., PPO, GRPO, hierarchical RL) | Which sub-actions or calls the agent chooses, and how it coordinates them |
| Ecosystem / runtime | The deployed environment where agents act and learn | Experience collection, reward shaping, orchestration | Datasets, trajectories, or policies for multiple agents |

## Recall: Fine tuning, in brief

Fine tuning refines a pre-trained model’s weights on domain-specific or task-specific examples to improve accuracy, style, or reasoning

- Collect (prompt → ideal response) pairs
- Train with gradient descent on supervised or RL objectives to update model weights
- Validate and deploy new model checkpoint

| **Type** | **Purpose** |
|---|---|
| Supervised fine-tuning (SFT) | Teach format, reasoning, tone |
| Instruction tuning | Align with human prompts |
| Domain tuning | Specialize to specific domains |
| LoRA / PEFT | Lightweight, adapter-based updates |

## Fine-tuning paradigms

| **Type** | **Description** | **Typical Use** |
|---|---|---|
| Supervised fine-tuning (SFT) | Train model on labeled examples of desired input→output | Instruction tuning |
| Reinforcement learning (RLHF/RLAIF) | Optimize model by reward feedback | Alignment |
| Agent fine-tuning | Optimize entire *agent workflow* using task success signals | Adaptive agents |

## Fine tuning, RL, agents

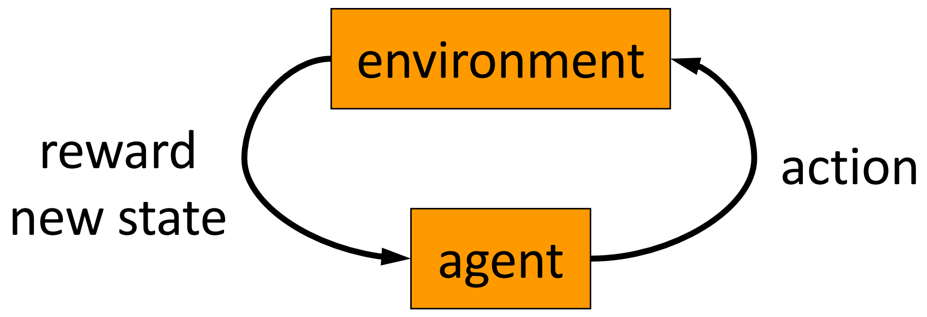
- **Fine-tuning** is a technique that changes the LLM itself.
  - Can be **supervised** (SFT, instruction tuning) or **reinforcement-based** (RLHF, GRPO, PPO)
  - Its output is an improved model checkpoint or adapter
- **Reinforcement Learning** is a *training paradigm* that can operate **inside or around** an agent
  - When the RL algorithm's gradient flows into the LLM weights → that's **RL-based fine-tuning**
  - When RL updates only the *policy logic* (e.g., planner, routing, parameter selection) → it's **agent-level training** without touching model weights.
- **Agent frameworks** (like Agent Lightning or ART) handle the *outer loop*: how experience is gathered, rewards computed, and updates applied.
  - Fine-tuning (of LLM weights) is one possible update target.
  - Prompt optimization, rule tuning, or memory shaping are others.

## Reinforcement learning

Reinforcement learning (RL), which has driven recent advances in reasoning models such as DeepSeek- R1 and Kimi k1.5, offers a powerful paradigm for optimizing LLMs in agentic scenarios. **While supervised learning requires detailed step-by-step annotations—which are scarce and costly for complex interactive tasks—RL relies on outcome-based reward signals**. This eliminates the need for task-specific curated data and allows agents to learn desirable behaviors directly from environment feedback across diverse tasks. Moreover, the trial-and-error nature of RL closely mirrors how humans acquire problem-solving skills, enabling models to learn action policies grounded in deployment contexts. This capability opens up the potential for transforming LLM-generated text tokens into real-world actions, making RL a natural fit for training models in agent-based systems.

## Reinforcement learning

- Problems involving an agent interacting with an environment, which provides numeric reward signals
- Goal: Learn how to take actions in order to maximize reward



# Fine tuning vs. reinforcement learning

| **When to Fine-Tune** | **When to Apply RL** |
|---|---|
| You have high-quality labeled data | Labels are unavailable but you can define a *reward* (success metric, correctness, user satisfaction) |
| You need consistent, static behavior (e.g., summarization style) | You need adaptive, goal-directed behavior (e.g., tool use, planning, dialogue). |
| Cost of annotation is lower than cost of rollout | Cost of environment interaction is lower than mass labeling |
| You want fast, repeatable training cycles. | You want continual improvement from real-world feedback |

## Why fine-tune LLMs?

LLMs trained on static Internet data struggle with real-world, interactive tasks, e.g.:

- Tool-using agents
- Code-executing or debugging agents
- Retrieval-augmented generation (RAG) agents
- Conversational agents in long-horizon interactions
- Scientific or experimental agents
- Game-playing or embodied agents

## Why static LLMs struggle in the real world

| **Domain** | **Example Failure** | **Why Static Data Fails** |
|---|---|---|
| Tool Use (SQL, APIs) | Misformats queries, can’t fix execution errors | Never sees real API responses or error messages |
| Code Agents | Outputs code that fails runtime tests | No reward for successful execution |
| Retrieval-Augmented QA | Retrieves irrelevant docs; hallucinates | No supervision from retrieval success/failure |
| Conversational Agents | Breaks down over long dialogs; repeats mistakes | No turn-level feedback or satisfaction signal |
| Scientific Agents | Suggests infeasible experiments | Never observes outcomes or experiment results |
| Embodied / Game Agents | Knows rules but can’t win | No experience-based learning from rewards |

## Why fine-tune agents?

LLMs trained on static Internet data struggle with real-world, interactive tasks

- Agents (tool users, planners, retrievers) generate **rich experience traces** unavailable in pretraining
- Real-world improvement loop:

  **Deploy → Observe → Reward → Update**

**Key idea**: *Environment provides the missing signal for continual learning*

## Tool-using agents: APIs, databases, …

**Example:** A text-to-SQL assistant like “Generate SQL for this query” works fine on benchmark data, but:

- fails when it encounters an unfamiliar schema or proprietary function (LEFT JOIN inventory vs. JOIN inv_table)
- does not know **when** to retry or **how** to parse an error message from a real database
- cannot adapt to reward signals like *execution success* or *query latency*

**Static-data limitation:** Training on example pairs does not expose the model to *feedback loops* or *action–outcome dynamics*. It never learns that *syntax errors* → *penalty, correct execution* → *reward.*

## Code-executing or debugging agents

**Example:** A code-writing model may output syntactically valid code but repeatedly fail runtime tests (e.g., off-by-one errors, undefined variables). As a static model it has no mechanism to:

- Re-run code, see failures, and adjust strategy
- Prefer code that passes tests over code that merely "looks right"

**Reinforcement Learning** (RL)-based adaptation can be a solution

- Training from *execution rewards* teaches it to explore, test, and self-correct: the idea behind **DeepSeek-R1** and **Agent Lightning's multi-step credit assignment**.

## Retrieval-augmented generation (RAG) agents

**Example:** A RAG system answering “What are the latest results on superconducting hydrides?” might:

- Retrieve irrelevant documents due to query mis-formulation
- Produce confident but hallucinated summaries

**Static LLMs fail because:**

- They’re trained to *predict text*, not to *optimize retrieval relevance or factuality*
- They have no gradient signal from “did this retrieval actually help answer the question?”

## Conversational agents in long-horizon interactions

- **Example:** A customer-service chatbot can generate fluent single replies, but breaks down when:
  - It needs to maintain consistent memory across 10–20 turns
  - It misinterprets user feedback (e.g., “That didn’t help”)
  - It can’t adapt its strategy after repeated failures
- **Static corpus problem:** No natural signal for *turn-level success*, *conversation satisfaction*, or *task completion*

## Scientific or experimental agents

- A chemistry-design agent proposes a synthesis plan that is infeasible when run in the lab, or doesn’t adjust when the experiment yields unexpected results
- Static pretraining lacks **closed-loop experience** with experimental outcomes; hence the push toward *real-world fine-tuning* from observed results

## Game-playing or embodied agents

- **Example:** An LLM describing “how to play chess” doesn’t learn *to win games*
- Winning requires trial-and-error reward feedback
- This is why systems like **OpenPipe/ART** and **Agent Lightning** treat the agent’s world as an *environment* with rewards

## Methods for correcting limitations of static data

| **Challenge** | **RL Solution** | **Example Framework** |
|---|---|---|
| No feedback loop | Collects environment rewards → updates model | **Agent Lightning, ART** |
| No credit assignment | Decomposes multi-turn traces into transitions | **LightningRL** hierarchical policy |
| No adaptation | Iteratively improves via rollout → reward → update | **OpenPipe / ART** pipelines |
| No intermediate signals | Automatic Intermediate Rewarding (AIR) | **Agent Lightning** client runtime |
| Long-horizon tasks | Policy learning across multiple steps | Hierarchical RL / GRPO / PPO |
| Static prompts | Optimizes prompts or examples from data | **DSPy** teleprompters |

## Summary

| **Challenge** | **Why static data fails** | **What RL fixes** |
|---|---|---|
| Action-dependent outcomes | No feedback loop in text corpora | Reward from environment outcomes |
| Error recovery | No notion of “try → fail → retry” | Credit assignment over sequences |
| Long-horizon consistency | Training truncates context | Policy learning across steps |
| Real-world variation | Internet text ≠ dynamic tools/APIs | Experience-driven adaptation |

## OpenPipe / ART: Agent Reinforcement Trainer

- **Core concepts**
  - “OpenPipe” is a middleware for reinforcement tuning of agents, providing interfaces for reward collection, logging, and evaluation
  - **ART (Agent Reinforcement Trainer)** abstracts away infrastructure, connecting LLMs, environments, and reward models
  - Supports multi-episode training, rollout–train cycles, and integration with RL frameworks like Hugging Face’s TRL or Microsoft’s VeRL
- **Key contribution:** makes RL-style fine-tuning *operational* for deployed agents, not just isolated models

## Agent Lightning: RL for any agent

- **Core ideas:**
  - **Training–Agent Disaggregation:** separates agent execution (client) from RL training (server)
  - **Unified Data Interface:** every LLM/tool call logged as (input, output, reward) transition.
  - **LightningRL:** hierarchical RL method compatible with PPO/GRPO/REINFORCE++ (no masking, no DAG parsing).
  - **Automatic Intermediate Rewarding (AIR):** converts system telemetry (e.g., tool success) into dense rewards.
- **Example:** training a calculator-using MathQA agent or text-to-SQL workflow; each tool call becomes a transition used for policy optimization.
- **Notes:**

  → Enables fine-tuning **without code modification** of existing agents (LangChain, AutoGen, etc.)

  → Bridges research RL frameworks (like VeRL) with real production agents.

## ART vs. Agent Lightning

| **Aspect** | **ART / OpenPipe** | **Agent Lightning** |
|---|---|---|
| Focus | Training *loop orchestration* and logging | Algorithm + data interface + system integration |
| Data model | Agent episodes with rewards | Unified `(state, action, reward)` transitions |
| Flexibility | Works with various RL frameworks | Works with *any* agent architecture |
| Innovation | Standardized RL infra for agents | Decoupled training–execution + AIR |

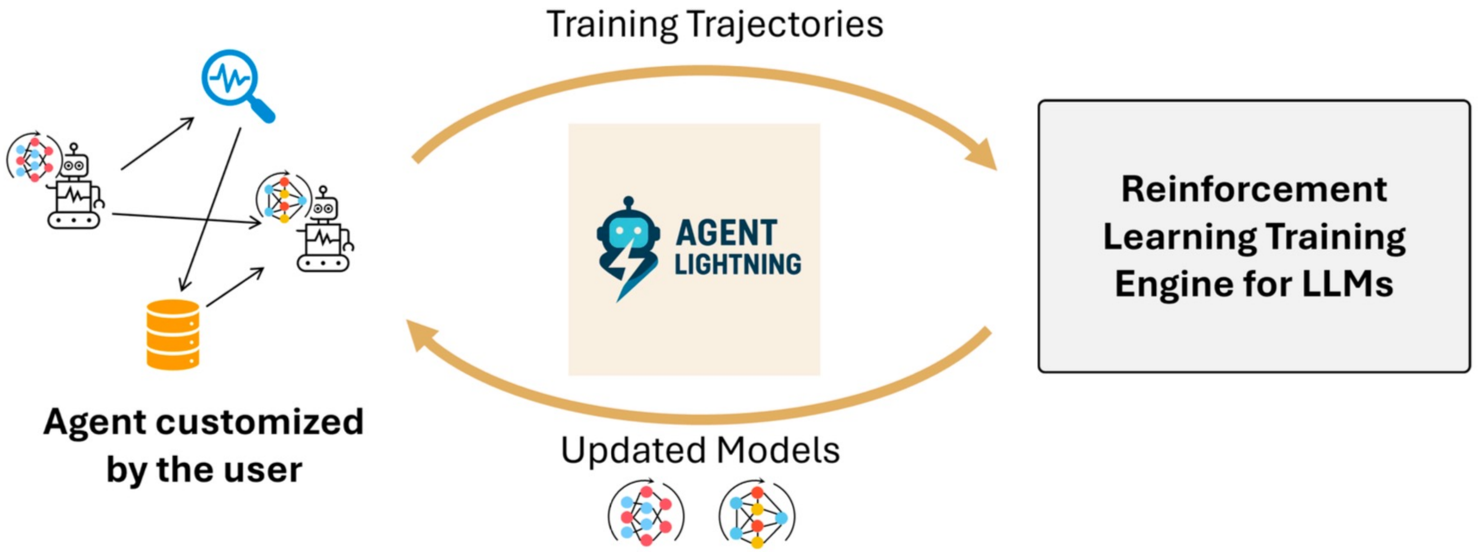

Training Trajectories
AGENT LIGHTNING
Reinforcement Learning Training Engine for LLMs
Agent customized by the user
Updated Models

Figure 1: Overview of *Agent Lightning*, a flexible and extensible framework that enables reinforcement learning of LLMs for ANY AI agents.

## For example, Math QA

**1) Algebraic Manipulation**

- **Problem:** If $x = 4$ and $y = 2x - 3$, compute $(x^2 - y^2)/(x - y)$.
- **Solution reasoning:**

$$y = 2(4) - 3 = 5$$
$$x^2 - y^2 = (16 - 25) = -9$$
$$x - y = -1$$

→ Result = 9

**2) Geometry / Trigonometry**

- **Problem:** A right triangle has sides of lengths 3 and 4. Find sine of larger acute angle
- **Solution reasoning:**

  Hypotenuse = 5

  sin(θ) = opposite/hypotenuse = 4/5 = **0.8**

**3)** "If $a = \frac{3}{5}$ and $b = 7$, compute $\left(a^{-2} + \sqrt{b}\right) - \dfrac{12}{3 \cdot 4}$. Return a single number."

## Agent Lightning and Math QA

- **Agent implementation:** Single-LLM workflow that (a) plans, (b) issues calculator calls, (c) integrates results, (d) answers
- **Tool:** calculator(expr: string) -> number (stateless).
  AIR (Automatic Intermediate Rewarding) will mark a tool call “valid/invalid” to yield intermediate rewards (format OK, tool executed, syntactically valid)
- **Terminal reward:** exact-match on numeric answer (1.0 if correct, else 0.0)
- **LightningRL** converts each **LLM call** into an **action**, does simple **credit assignment** (same return to each action by default), then applies a single-turn RL loss (e.g., GRPO/PPO) token-wise on each call’s output

## Ground-truth answer (for reference)

"If $a = \frac{3}{5}$ and $b = 7$, compute $\left(a^{-2} + \sqrt{b}\right) - \frac{12}{3 \cdot 4}$. Return a single number."

- Compute stepwise (what the **calculator** is for):
  - $a^{-2} = \left(\frac{3}{5}\right)^{-2} = \left(\frac{5}{3}\right)^{2} = \frac{25}{9}$
  - $\sqrt{b} = \sqrt{7}$
  - $\frac{12}{3 \cdot 4} = \frac{12}{12} = 1$
- So the target is:

$$\frac{25}{9} + \sqrt{7} - 1 = \frac{16}{9} + \sqrt{7} \approx 1.777\ldots + 2.64575\ldots \approx 4.42275$$

## Agent execution timeline

- **Transition T1 – Plan & Compute (3/5)^(-2)**
  LLM output: {"expr":"(3/5)^(-2)"}
  Tool: 2.7778
  AIR reward ≈ 0.15
- **Transition T2 – Compute sqrt(7)**
  Tool: 2.6458
  AIR reward ≈ 0.15
- **Transition T3 – Compute 12/(3*4)**
  Tool: 1
  AIR reward ≈ 0.15
- **Transition T4 – Combine & Answer**
  LLM: {"answer":"(25/9)+sqrt(7)-1≈4.4228"}
  Terminal reward = 1.0 (correct)
- **Total Return R ≈ 1.45**

Automatic Intermediate Rewarding (AIR) enables the assignment of intermediate rewards to transitions based on system monitoring signals (such as tool call return statuses)

## T1 computation in more detail

- Input_1 (state → observation):

  **System**: You are a math solver. Use the calculator tool for exact arithmetic.

  **User**: If a=3/5 and b=7, compute (a^-2 + sqrt(b)) - 12/(3*4).

  Respond in JSON with fields: {"plan": "...", "next_action": "compute" | "answer", "expr": "..."}

- LLM Output_1 (action a1):

  {"plan":"Compute a^-2, compute sqrt(7), compute 12/(3*4), then combine.", "next_action":"compute","expr":"(3/5)^(-2)"}

- AIR intermediate reward r_1 (format): +0.05 if JSON parseable & fields present
- AIR intermediate reward r_1 (tool-eligibility): +0.05 if expr is calculator-valid (simple static check)

## Unified data trace

- **Logged transitions (simplified JSONL):**

```
{"t":1, "input":"UserInput", "output":"(3/5)^(-2)", "reward":0.15}
{"t":2, "input":"...ToolResult1...", "output":"sqrt(7)", "reward":0.15}
{"t":3, "input":"...ToolResult2...", "output":"12/(3*4)", "reward":0.15}
{"t":4, "input":"...ToolResult3...", "output":"Answer≈4.4228", "reward":1.0}
```

- **Observations**
  - Each LLM call = one action in the Markov decision process
  - No masking, no concatenation: just clean transitions for RL

## LightningRL Optimization Flow

- **Credit Assignment:** Assign per-step or uniform return to each transition
- **Token-Level Optimization:** Apply single-turn RL loss (GRPO/ PPO/ REINFORCE++)
- **Batching:** Transitions grouped by task for advantage estimation
- **AIR:** Provides dense shaping rewards to accelerate learning
- **Benefits:**
  - Modular and scalable (no coupling between agent logic & RL engine)
  - Works across AutoGen, LangChain, or custom agents

## Policy-gradient RL for language models

- **Policy-gradient** methods optimize a parameterized model $\pi_{\text{theta}}$(a | s) (the LLM) to *increase expected reward*. The general objective is:

$$\nabla_\theta J(\theta) = \mathbb{E}_{s,a\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta\,(a \mid s)\, A(s, a)\right],$$

  where **A(s,a)** ("advantage") measures how much better the sampled action's reward is than average

- In LLM fine-tuning,
  - **state s** = prompt or context
  - **action a** = the generated text
  - **reward r** = numeric score (helpfulness, correctness, etc.)
  - **advantage A** = signal telling the model which responses were better

## Policy gradient methods used in Agent Lightning

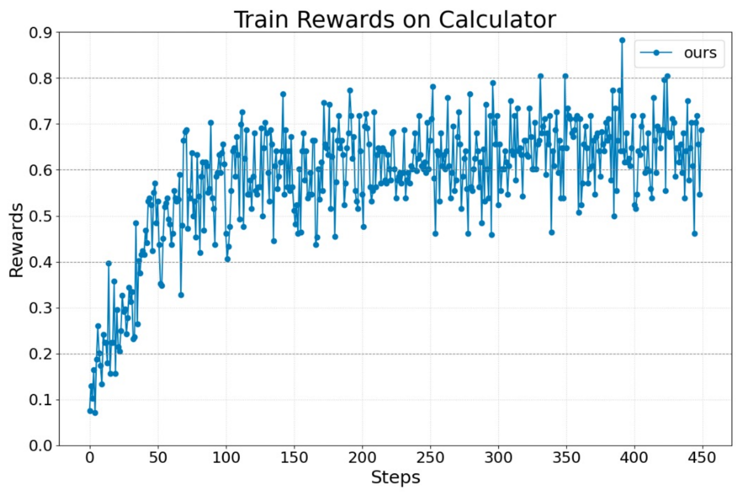
- **Proximal Policy Optimization (PPO):** take conservative updates so new policy does not drift too far from previous
- **Group Relative Policy Optimization (GRPO):** Group several model outputs for same prompt/task, normalize rewards. Default.
- **REINFORCE++:** Simplest; no critic, no grouping

Methods differ only in **how they compute A(s,a)**

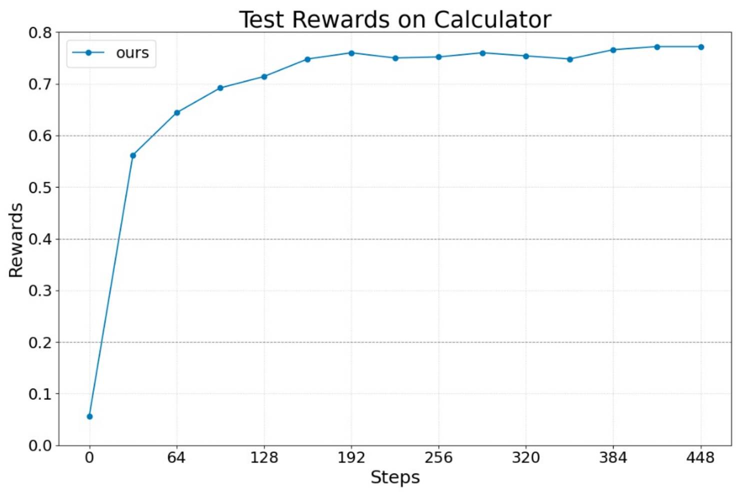
| **Feature** | **PPO** | **GRPO** | **REINFORCE++** |
|---|---|---|---|
| Critic network | yes | none | none |
| Advantage baseline | Value function | Group mean / std | Batch mean |
| Stability | Very high | Moderate | Lower |
| Compute cost | Highest | Medium | Lowest |
| Use in Agent Lightning | Supported but heavy | Default choice | Optional |

## Training results

- **Dataset:** Calc-X + **Base Model:** Llama-3.2-3B-Instruct
- Smooth, consistent improvement in both train & test reward curves
- Improved accuracy in symbolic + numeric tasks
- Robust handling of multi-turn reasoning with tool invocations



(a) Train reward



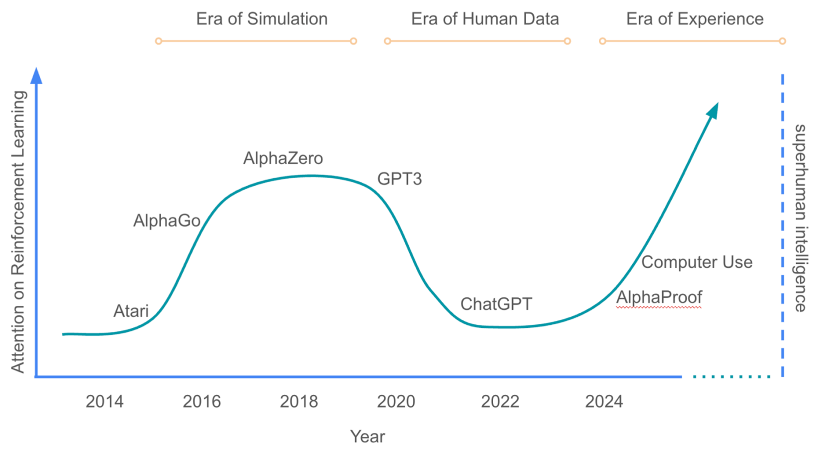
(b) Test reward

## Agent Lightning summary

- **Transition-based modeling** enables fine-grained RL on complex workflows
- **Automatic Intermediate Rewarding (AIR)** mitigates sparse reward problem
- **LightningRL** reuses efficient single-turn RL across multi-step, tool-augmented agents
- **Training-Agent Disaggregation** (server $\leftrightarrow$ client) allows zero agent code modification
- **Outcome:** Agent Lightning continuously improves tool-using math agents, achieving both reliability and scalability

# Welcome to the Era of Experience

David Silver, Richard S. Sutton*



Our contention is that incredible new capabilities will arise once the full potential of experiential learning is harnessed. This era of experience will likely be characterised by agents and environments that, in addition to learning from vast quantities of experiential data, will break through the limitations of human-centric AI systems in several further dimensions:

- Agents will inhabit streams of experience, rather than short snippets of interaction.
- Their actions and observations will be richly grounded in the environment, rather than interacting via human dialogue alone.
- Their rewards will be grounded in their experience of the environment, rather than coming from human prejudgement.
- They will plan and/or reason about experience, rather than reasoning solely in human terms