Contents

1 Teaching an LLM to Explore: Reinforcement Learning for Document Navigation

How we trained a Qwen3-8B model to efficiently search through documents using GRPO, inspired by Karpathy's autoresearch — and what Goodhart's Law taught us along the way.

1.1 Introduction — The Autoresearch Connection

In January 2025, Andrej Karpathy announced autoresearch — a project where an AI agent autonomously improves a language model by searching over code edits, running experiments within a 5-minute budget, and optimizing validation perplexity. The core insight was simple but profound: **give an agent a search space, a budget, and a scalar reward, then let reinforcement learning do the rest.**

Karpathy's search space was code modifications. Ours is something different: **document exploration**.

We built **Tinker-Explorer**, an RL agent that learns to navigate a set of document chunks to answer multi-hop questions. Like autoresearch, it operates under a budget (limited steps), must decide what information to gather (which documents to read), and receives a scalar reward (answer correctness). Unlike autoresearch, the search space isn't code — it's evidence.

The mapping between the two projects:

	Karpathy's Autoresearch	Tinker-Explorer
Agent	Code-editing LLM	Document-exploring LLM
Search space	Code modifications	Document chunk selections
Budget	5 minutes wall-clock	10 action steps max
Reward	val_bpb improvement	Token F1 on answer
Optimization	RL over code edits	GRPO over exploration trajectories
Key challenge	Which edits improve the model?	Which documents contain the answer?

Both projects share the same fundamental question: *Can an LLM learn to make better decisions about what to explore, through trial and error?*

This post documents our journey across three training runs — including a Goodhart's Law failure, a reward debugging mystery, and the realization that looking at your model's actual outputs matters more than tuning hyperparameters.

1.2 The Problem: Partially Observable Multi-Hop QA

Standard QA gives the model everything upfront. Retrieval-augmented QA retrieves documents first, then answers. But neither captures the **active information gathering** that humans do

naturally.

Consider this question:

"Which film has the director born first, Once A Gentleman or The Girl In White?"

A human researcher would:

1. Look at the available sources
2. Open the article about "Once A Gentleman" to find the director's birth year
3. Open the article about "The Girl In White" to do the same
4. Compare the two dates and answer

This process requires **sequential decision-making under uncertainty** — the agent doesn't know which documents are useful until it reads them. This is fundamentally a reinforcement learning problem.

1.2.1 Why Not Just Retrieve Everything?

You could argue: "Just open all the documents." But in real-world settings:

- API calls cost money (think: calling a paid knowledge base)
- Context windows have limits
- Irrelevant information introduces noise that degrades LLM performance
- Some documents are red herrings that actively mislead

The goal isn't just to answer correctly — it's to answer correctly **while reading as few documents as possible**. This is the efficiency-accuracy tradeoff that makes the problem interesting.

1.3 Environment Design

Tinker-Explorer implements a standard RL environment loop:

1.3.1 State Space

At each step, the agent observes:

- **The question** (e.g., "Who is Marie Zéphyrine Of France's paternal grandmother?")
- **A list of chunk previews** — one-line titles of all available documents, but NOT their contents
- **Previously opened chunks** — full text of any documents the agent has chosen to read
- **Remaining step budget** — how many actions it has left

1.3.2 Action Space

Three possible actions at each step:

```
{"action": "OPEN", "target": 3, "reasoning": "Chunk 3 mentions Marie Zéphyrine..."}
{"action": "SUMMARIZE", "target": 3, "text": "Born 1750, daughter of Louis XV..."}
{"action": "ANSWER", "text": "Marie Leszczyńska"}
```

- **OPEN(i)**: Read chunk `i`'s full text. This is the exploration action.
- **SUMMARIZE(i)**: Write a summary of chunk `i`. This helps manage context length.
- **ANSWER(text)**: Submit a final answer. Ends the episode.

1.3.3 Reward Function

The agent receives reward only when it answers. The reward is the **token-level F1 score** between its predicted answer and the gold answer — a standard metric from the SQuAD literature that gives partial credit for overlapping words.

For example:

- Gold: "The Mask Of Fu Manchu" → Predicted: "The Mask of Fu Manchu" → F1 = 1.0 ✓
- Gold: "Małgorzata Braunek" → Predicted: "Chunk 4 has been opened" → F1 = 0.0 ✗

1.3.4 Episode Flow

```
1. Agent receives question + chunk previews
2. Agent decides: OPEN, SUMMARIZE, or ANSWER?
3. If OPEN/SUMMARIZE → environment reveals text, step counter increments
4. If ANSWER → episode ends, reward = token_f1(predicted, gold)
5. If step budget exhausted → episode ends with reward = 0
```

A typical successful episode takes 2-4 steps: open the relevant chunks, then answer.

1.4 The Dataset: 2WikiMultiHopQA

We use 2WikiMultiHopQA — a multi-hop question answering dataset built from Wikipedia. Each question requires reasoning across exactly two Wikipedia articles.

Why this dataset?

1. **Natural chunk structure**: Each Wikipedia article is a chunk. The agent must decide which articles to read.

2. **Ground truth supporting facts**: The dataset provides `supporting_facts` — which paragraphs contain the answer evidence. This lets us measure whether the agent opens the *right* documents.
3. **Multi-hop reasoning**: Questions require combining information from two sources. The agent can't answer from a single document.
4. **Varied question types**: Comparison ("which film came first"), bridge ("who is the mother of the director of..."), and compositional questions.

1.4.1 Dataset Split

- **Training**: 5,000 tasks (randomly sampled from the full 167K)
- **Validation**: 200 held-out tasks for evaluation
- **Chunk pool**: Each task has ~10 candidate chunks (2 relevant, ~8 distractors)

1.5 Training Infrastructure: Tinker + GRPO

1.5.1 Tinker Platform

All training runs on Tinker (Temporal Intelligence via Neural Knowledge Extraction and Reasoning) — a cloud platform that provides:

- **Hosted model weights** — Qwen3-8B with LoRA adapters, no local GPU needed
- **Sampling API** — generate completions from the current policy
- **Training API** — submit gradient updates (importance-sampled policy gradient)
- **Checkpointing** — save/restore model states

This architecture is powerful: the model lives in the cloud, and our local code just orchestrates rollouts and computes gradients. We ran all experiments on a MacBook with no GPU.

1.5.2 GRPO (Group Relative Policy Optimization)

We use **GRPO** — a variant of policy optimization from DeepSeek-R1 — instead of PPO. The key idea:

1. For each task, sample `G = 16` trajectories from the current policy
2. Compute rewards for all 16
3. Normalize rewards within the group: `advantage_i = (r_i - mean(r)) / std(r)`
4. Update the policy to increase probability of above-average trajectories

Why GRPO over PPO?

- **No value function needed** — PPO requires a separate critic network. GRPO computes baselines from the group.
- **Natural for language** — each trajectory is a sequence of tokens. GRPO treats the whole sequence as one "action."
- **Simpler implementation** — just a weighted policy gradient update, no GAE, no clipping ratio (handled by importance sampling).

1.5.3 Hyperparameters

Parameter	Value	Rationale
GROUP_SIZE	16	Balance between variance reduction and compute
BATCH_SIZE	8	Tasks per batch (= 128 total rollouts per update)
Learning Rate	5e-6	Conservative — RL fine-tuning is sensitive
LoRA Rank	32	Enough capacity without overparameterizing
Grad Clip Norm	1.0	Prevents IS loss explosions
Max Steps	10	Episode timeout

1.5.4 SFT Warm-Start

Before RL training, we performed supervised fine-tuning (SFT) on 450 demonstration episodes generated by a heuristic policy. This gives the model a reasonable starting point — it knows the action format and basic exploration strategy.

The heuristic policy simply:

1. Computes TF-IDF overlap between the question and each chunk title
2. Opens the highest-overlap chunk
3. Answers with the chunk title (which is often the Wikipedia article title — and therefore the exact entity name)

This heuristic achieves **F1 = 0.246** — a surprisingly strong baseline.

1.6 Run 1: Shaped Reward — The Goodhart Trap

1.6.1 Hypothesis

"If we add a bonus for opening relevant chunks, the model will learn to explore more effectively."

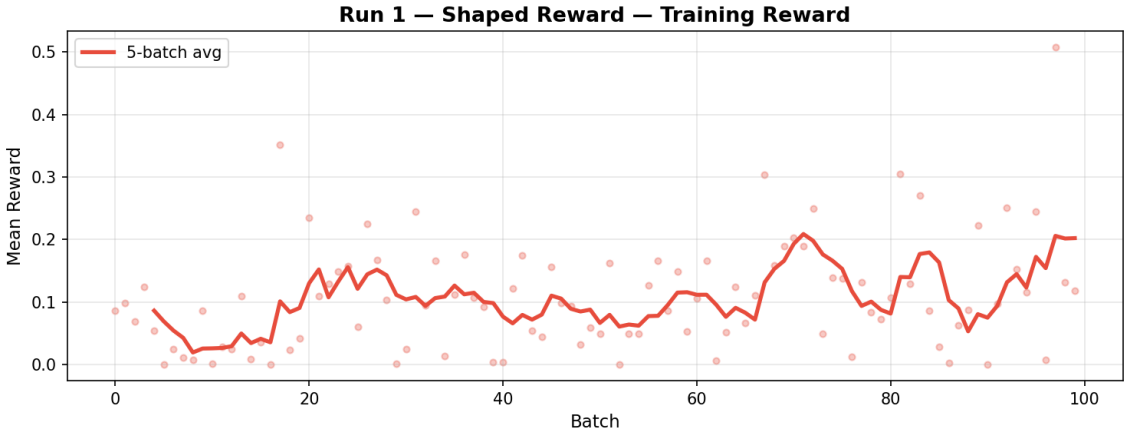
1.6.2 Reward Design

```
reward = token_f1(predicted, gold) + 0.2 × (number of relevant chunks opened)
```

The idea was sound: reward the agent not just for the final answer, but for the intermediate exploration steps. Every relevant chunk opened earns a +0.2 bonus (capped at 0.4).

1.6.3 Training

100 batches over ~12 hours. Training reward climbed steadily — the 5-batch average increased from 0.08 to 0.20 over the course of training.



1.6.4 Results

Model	F1	EM	Answer Rate
Heuristic	0.246	0.220	100%
SFT	0.152	0.065	87%
RL Run 1	**0.123**	**0.010**	96%

Wait — F1 went DOWN? The RL agent performed *worse* than the SFT baseline it started from. Training reward increased, but actual answer quality decreased.

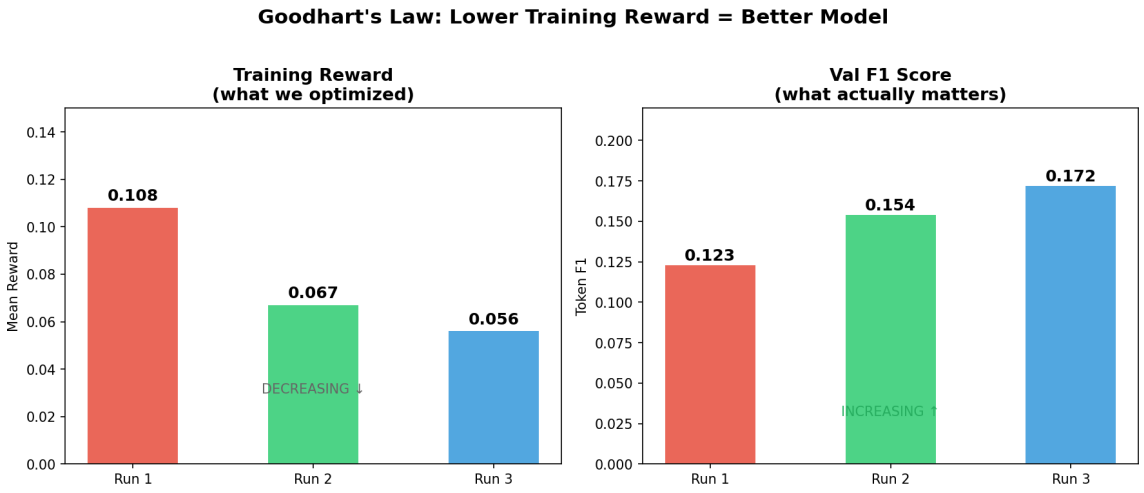
1.6.5 The Diagnosis: Goodhart's Law

> "When a measure becomes a target, it ceases to be a good measure."

The relevance bonus created a shortcut: the agent learned to open the right chunks (earning the +0.2 bonus) while paying less attention to actually answering correctly. The training reward was

dominated by the exploration bonus, not answer quality — so the gradient signal pushed the model to optimize opens, not answers.

This is a textbook case of **reward hacking**. The agent found a way to earn high reward without doing the thing we actually wanted (accurate answers).



1.6.6 Lesson

Shaped rewards are dangerous. The extra signal might seem helpful, but if it's easier to optimize than the true objective, the agent will chase the bonus and ignore what matters.

1.7 Run 2: Pure F1 — Cleaning the Signal

1.7.1 The Fix

Strip everything back to the minimum:

```
reward = token_f1(predicted, gold)    # nothing else
```

Plus one safety gate: the agent must open at least 1 chunk before earning any reward. This prevents the degenerate strategy of answering immediately without reading anything.

1.7.2 Training

100 batches. Immediately obvious: training reward was **lower** than Run 1. The agent earned less reward per batch because there was no easy bonus to collect.

The red line (Run 1) sits higher than the green line (Run 2) throughout training. To someone only watching the training curves, Run 1 looks better. But...

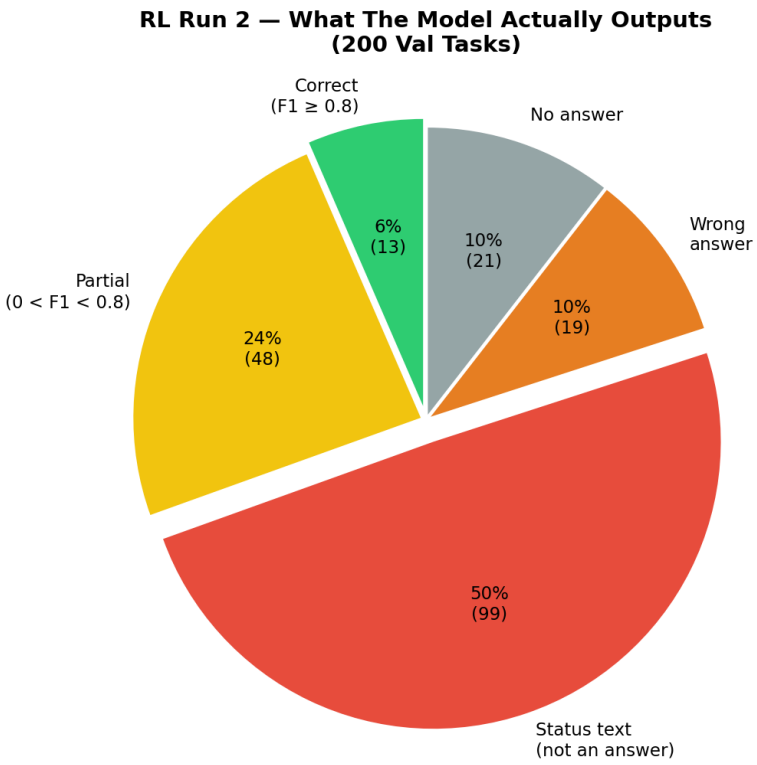
1.7.3 Results

Model	F1	EM	Answer Rate
Heuristic	0.246	0.220	100%
SFT	0.152	0.065	87%
RL Run 1 (shaped)	0.123	0.010	96%
RL Run 2 (pure F1)	**0.154**	**0.065**	89.5%

F1 jumped from 0.123 to 0.154 — a 25% improvement — by *removing* reward signal. The pure F1 agent matched the SFT baseline, which means the RL training at least didn't hurt, even if it didn't help much.

1.7.4 Error Analysis

We looked at what the model actually output on the 200 val tasks:



50% of outputs were status text — things like:

- "Chunk(s) 4 and 7 ('Polish-Russian War' and 'Xawery Żuławski') have been opened."
- "Chunk 0 ('Blind Shaft') has highest overlap with the question."

The model was outputting **descriptions of its own actions** instead of answers. It opened the right chunks, formed the right reasoning — then put the reasoning in the answer field instead of the actual answer.

This error mode is invisible in aggregate metrics. You can only find it by reading examples.

1.8 Run 3: Status Text Fix — The Breakthrough

1.8.1 The Insight

The 50% status text rate meant that half our training signal was wasted. These episodes generated zero reward (because "Chunk 4 has been opened" has zero token overlap with "Małgorzata Braunek") — but the model didn't learn from them because the gradient signal was the same as for genuinely wrong answers.

We needed to make the punishment *explicit*.

1.8.2 Three Changes

1. Reward penalty — If the answer starts with known status text patterns, force reward to 0:

```
_STATUS_PREFIXES = ("chunk", "the chunk", "chunks", "i have", "i've", "based on")

if any(answer_text.lower().startswith(p) for p in _STATUS_PREFIXES):
    return 0.0  # you gave me reasoning, not an answer
```

2. Prompt update — Added an explicit "NEVER do this" section:

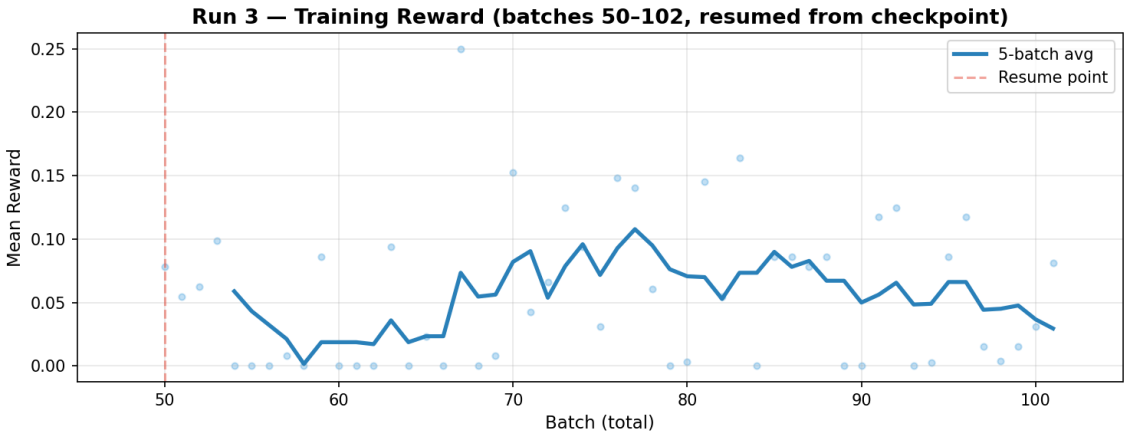
```
NEVER put any of these in the "text" field:
- "Chunk X has been opened" - this is NOT an answer
- "Chunk X has highest overlap" - this is reasoning, not an answer
- "Based on the text..." - just give the answer directly
```

3. Same pure F1 base — No relevance bonus. Min-read gate still active.

1.8.3 Training

~102 batches across two sessions (the first session crashed at batch 52 due to a Tinker API network timeout; we resumed from the batch 50 checkpoint).

Training reward was the **lowest of all three runs** — the status text penalty made the signal even sparser (71% nonzero batches, vs 78% in Run 2 and 96% in Run 1).

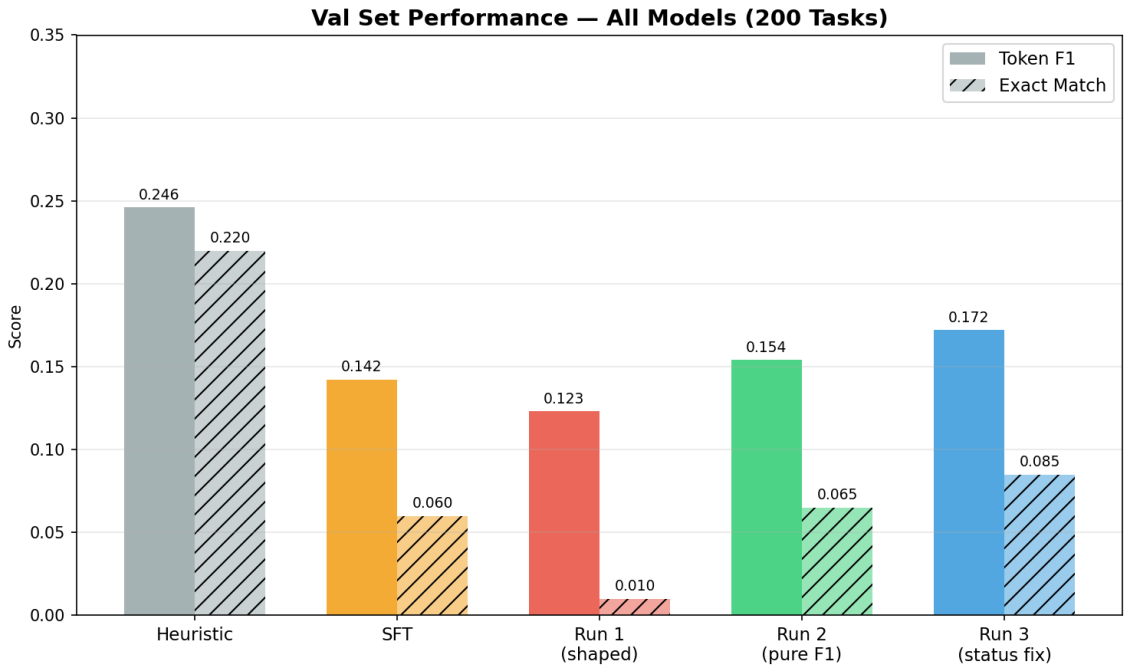


1.8.4 Results

Model	F1	EM	Answer Rate	Avg Opens
Heuristic	**0.246**	**0.220**	100%	1.2
SFT	0.142	0.060	87.0%	1.6
RL Run 1 (shaped)	0.123	0.010	96.0%	1.4
RL Run 2 (pure F1)	0.154	0.065	89.5%	1.5
RL Run 3 (status fix)	**0.172**	**0.085**	**99.5%**	**1.2**

Best results across every metric:

- **F1 = 0.172** — 40% better than Run 1, 21% better than SFT
- **EM = 0.085** — 8.5× better than Run 1
- **Answer Rate = 99.5%** — the model almost always gives an answer now
- **Efficiency = 1.2 opens** — same as the heuristic, better than SFT



1.8.5 The Reasoning Evolution

The most compelling evidence comes from looking at the same questions across all three runs:

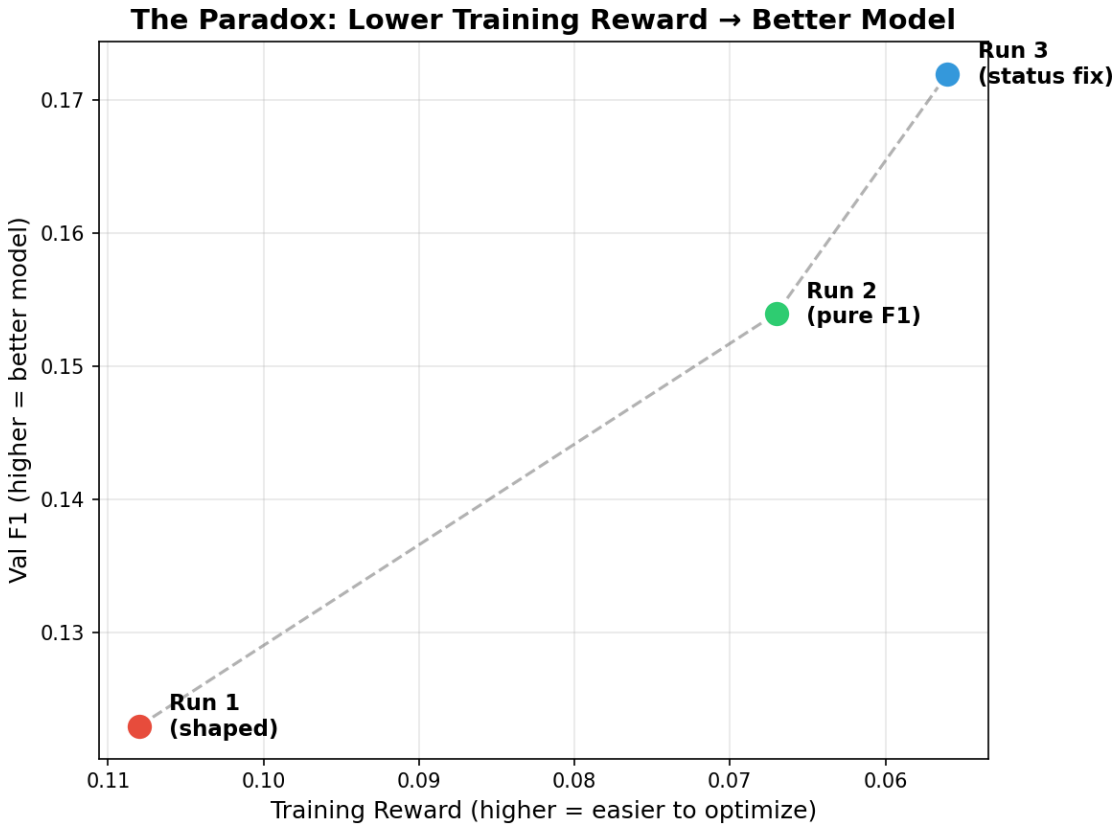
Example 2 tells the whole story:

- **Run 1**: "Chunk(s) 4 and 7 have been opened." → F1 = 0.00
- **Run 2**: "Chunk(s) 4 and 7 have been opened." → F1 = 0.00
- **Run 3**: "Małgorzata Braunek" → F1 = 1.00

The model opened the right documents in all three runs. The reasoning was correct in all three runs. The only difference was **what it put in the answer field**.

1.9 The Paradox: Lower Training Reward = Better Model

This is the central insight of the project. Across three runs, we observed an inverse relationship between training reward and actual model quality:



Run	Training Reward	Val F1
Run 1 (shaped)	**0.108** (highest)	**0.123** (worst)
Run 2 (pure F1)	0.067	0.154
Run 3 (status fix)	**0.056** (lowest)	**0.172** (best)

The run with the lowest training reward produced the best model.

Why? Because each successive run used a **stricter, more honest reward function**:

- Run 1's reward was easy to earn (bonus for opens) but misleading
- Run 2's reward was harder (F1 only) but still didn't penalize status text

- Run 3's reward was the hardest (F1 + status text penalty) but most aligned with what we actually want

Each time we made the reward harder, the training curves looked worse — but the model learned better behaviors. This is the opposite of what intuition suggests. Most practitioners assume higher training reward = better learning. Our experience shows that **reward quality matters more than reward quantity**.

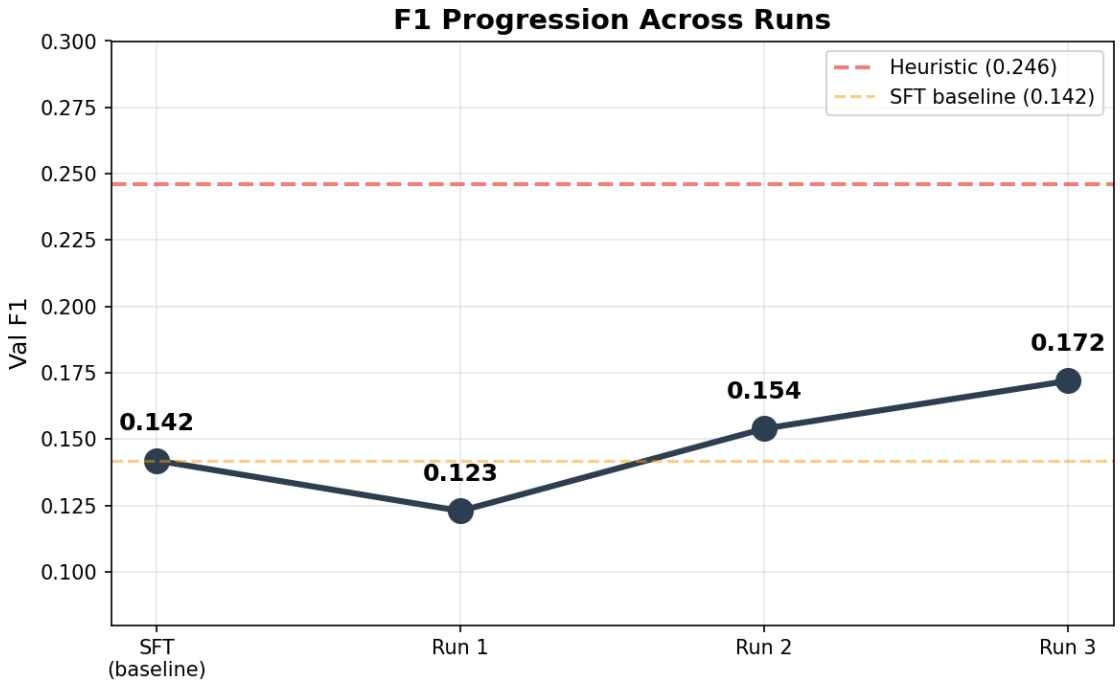
This echoes a broader principle in RL: **easy rewards lead to shallow learning**. The difficulty of the reward function is a feature, not a bug.

1.10 Results and Analysis

1.10.1 Final Comparison Table

Model	Token F1	Exact Match	Answer Rate	Avg Opens	Training
Heuristic	**0.246**	**0.220**	100%	1.2	N/A
SFT (warm-start)	0.142	0.060	87.0%	1.6	450 steps
RL Run 1 (shaped)	0.123	0.010	96.0%	1.4	100 batches
RL Run 2 (pure F1)	0.154	0.065	89.5%	1.5	100 batches
RL Run 3 (status fix)	**0.172**	**0.085**	**99.5%**	**1.2**	~102 batches

1.10.2 The Improvement Trajectory



Starting from the SFT baseline (0.142), RL initially made things worse (Run 1: 0.123), then systematically improved through reward engineering (Run 2: 0.154, Run 3: 0.172). The gap to the heuristic (0.246) narrowed from 50% to 30%.

1.10.3 Answer Quality Distribution

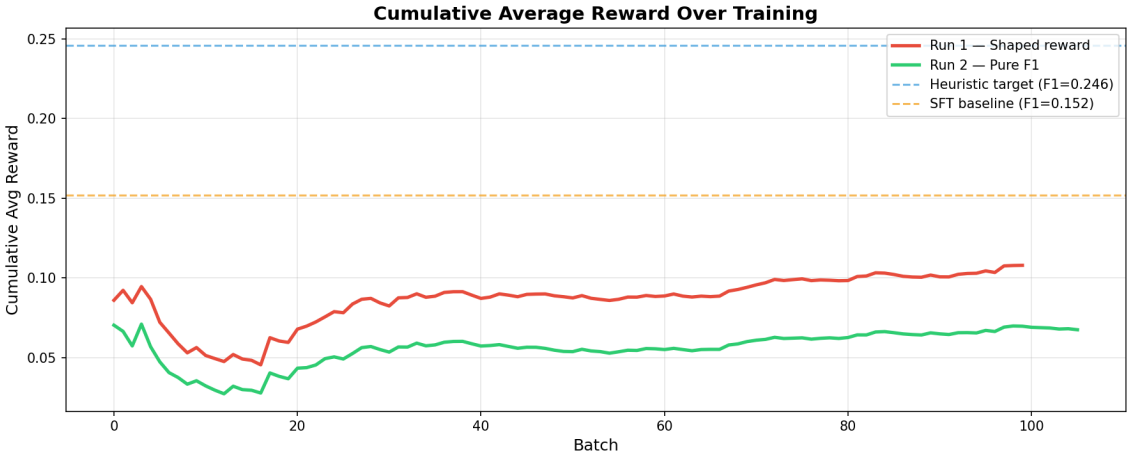
The most dramatic change is in the "Status text" category — dropping from ~120 instances in Run 1 to significantly fewer in Run 3, while "Correct" and "Partial" answers increased.

1.10.4 Where the Gap to the Heuristic Remains

The heuristic still leads by 30%. Why?

1. **Exact entity names**: The heuristic uses Wikipedia article titles as answers, which happen to be the exact entity names that 2WikiMultiHopQA expects. The RL model generates free-text that may paraphrase or include minor variations.
2. **Formatting**: "The Mask Of Fu Manchu" vs "The Mask of Fu Manchu" — capitalization differences reduce F1.
3. **Inherent difficulty**: Some questions require reasoning chains the 8B model struggles with, regardless of which documents it reads.

1.10.5 Cumulative Reward



The cumulative average reward shows both RL runs plateauing well below the heuristic and SFT baselines — confirming that the training signal, while sufficient for improvement, remains sparse.

1.11 Lessons Learned

1.11.1 Look at Your Model's Actual Outputs

This is the single most important lesson. We spent hours tuning hyperparameters and reward functions — but the breakthrough came from **reading 20 example predictions** and noticing that half were status text. No amount of quantitative analysis would have revealed this.

Practical advice: Before any training run, sample 20 predictions and read them manually. Two minutes of qualitative analysis beats two hours of hyperparameter sweeps.

1.11.2 Goodhart's Law Is Real and Painful

Shaped rewards feel scientific and principled. Adding a bonus for opening relevant chunks seems like it should help. In practice, it creates a shortcut the agent exploits while ignoring the actual objective.

Practical advice: Start with the simplest possible reward. Add complexity only when you've confirmed the base signal works.

1.11.3 Sparse but Honest > Dense but Misleading

Run 3 had 71% nonzero batches (vs 96% for Run 1). Many practitioners would look at that and add reward shaping to "help" the agent learn. Our experience shows the opposite: the sparse signal produced the best model because every bit of reward was earned through genuinely correct behavior.

1.11.4 SFT Warm-Start Matters More Than You Think

Without SFT, the model wouldn't know the action format (JSON with "action", "target", "text" fields). The 450 demonstration episodes gave it the basic vocabulary for exploration. RL then refined the *decisions* within that format.

1.11.5 Infrastructure Resilience is a First-Class Concern

Across three runs (~300 total batches), we experienced:

- 2 network timeouts requiring session restart
- 1 tensor alignment bug requiring code fix
- Multiple loss spikes from importance sampling ratios

Long-running RL experiments need: checkpointing every N batches, automatic resume logic, and gradient clipping. These aren't optional — they're survival features.

1.11.6 The Heuristic Baseline Was Shockingly Strong

A simple TF-IDF overlap heuristic achieved F1 = 0.246. After 3 RL runs totaling ~300 batches (~36 hours of training), the learned agent reached 0.172. The heuristic required zero training.

This humbling result is common in RL: carefully engineered baselines are hard to beat. The heuristic's advantage came from a design choice (using chunk titles as answers) that happened to align perfectly with the evaluation metric. The RL agent had to discover this alignment from scratch.

1.12 What's Next

The 30% gap to the heuristic is surmountable. Based on our error analysis, these are the highest-ROI next steps:

1.12.1 Near-Term (Runs 4-5)

1. **Exact Match reward** — Replace token F1 with binary EM. The heuristic wins because its answers are exact matches. Training for EM directly would close the gap, though the sparser signal needs a larger GROUP_SIZE (32 or 64).
2. **Post-processing cleanup** — Strip common prefixes ("The answer is...", "Based on the text...") at eval time. This is a free 2-3% F1 improvement.
3. **Few-shot prompting** — Add 3-4 worked examples to the system prompt showing the complete question → open → answer flow.

1.12.2 Medium-Term

4. **Larger GROUP_SIZE** (32-64) — More rollouts per task reduce variance, which is critical with sparser rewards.
5. **Curriculum learning** — Start with easier questions (single-hop) and gradually introduce multi-hop. This provides denser early reward without sacrificing eventual difficulty.
6. **SEARCH(q) sub-action** — Let the agent compose sub-queries. Instead of just opening chunks by index, it could search for "director of Polish-Russian War" and get filtered results.

1.12.3 Long-Term

7. **Larger models** — Qwen3-8B is capable but limited. 14B or 32B models may have stronger reasoning that translates to better exploration.
8. **Multi-hop reasoning chains** — Visualize and reward intermediate reasoning steps, not just the final answer.

1.13 Appendix: Technical Details

1.13.1 Code Structure

```
tinker-explorer/
|-- env/
|   |-- explorer_env.py      # RL environment
|   |-- action_schema.py     # JSON action parsing
|   `-- reward.py            # Reward function (all 3 variants)
|-- train/
|   `-- rl_train.py          # GRPO training loop
|-- eval/
|   |-- eval_rl.py           # Val set evaluation
|   `-- eval_trajectories.py # Trajectory visualization
|-- policies/
|   |-- prompts.py           # System prompt
|   `-- heuristics.py        # Heuristic baseline
|-- runs/                    # Run reports (this post's source data)
|-- plots/                   # All figures in this post
`-- logs/                    # Raw metrics and rollouts
```

1.13.2 Compute

All experiments ran on a MacBook Pro with no local GPU. Model inference and gradient computation happened on Tinker's cloud infrastructure. Total wall-clock time across 3 runs: ~40 hours. Estimated cloud compute: ~120 GPU-hours (Qwen3-8B with LoRA on A100-equivalent).

1.13.3 Reproducibility

All metrics, checkpoints, and evaluation results are saved in the repository:

- `logs/rl-run-v{1,2,3}/metrics.jsonl` — per-batch training metrics
- `logs/eval-rl-v1.json` — full per-task evaluation results
- `runs/run_{1,2,3}_*.md` — detailed run reports with interpretation

This project was built as a research experiment exploring RL for language agent training. The code, data, and this post are available for educational purposes. Inspired by Karpathy's autoresearch , built on Tinker .