Extending the project to perform both tool calls as well as analyzing the responses based on the tool outputs

Short answer first: **yes**, this makes mathematical sense and **yes**, it's fully compatible with SkyRL.

- In RL terms, you're defining a **dense per-turn reward** for tool usage + analysis **and** a **terminal reward** that scores the **final text answer** against a **reference answer** produced by your synthetic data generator. Maximizing expected return under GRPO/PPO will push the policy toward correct tools/arguments *and* high-quality final summaries.
- SkyRL is built for this: multi-turn `BaseTextEnv` tasks, tool calls via ToolGroups, final answers (e.g., the Search example uses `<answer>…</answer>` and gives reward for correctness), and dataset entries that keep only a seed prompt while the environment computes turn-level rewards. (skyrl.readthedocs.io)

Below is a **complete recipe**:

---

0) What "mathematically makes sense" here

Let an episode be $\tau = (s_0,a_0,r_0,\dots,s_T,a_T,r_T)$ where:

- Steps $1..K$ are **tool turns**; step $K{+}1$ is the **final text answer**.
- Reward $r_t = r_t^{\text{tools}}$ for tool turns (shaped by your rubric), and $r_{K+1} = \alpha \cdot r^{\text{final}}$ where $r^{\text{final}} \in [0,1]$ comes from **LLM-as-a-Judge** (LAJ) comparing the **policy's final answer** vs the **reference** (and/or a set of "verifiable facts" extracted during data generation). α balances tool learning vs answer quality.

GRPO/PPO maximizes $\mathbb{E}_\pi\left[\sum_t r_t\right]$ using advantages A_t (e.g., GAE). Since LAJ outputs are **numeric rewards**, they're just another term in the return; no special math is needed. (This is analogous to SkyRL's Search example, which assigns a terminal 0/1 for correct/incorrect final text. We're just using a **graded** judge instead of binary correctness.) (skyrl.readthedocs.io)

---

1) Data model you should write to `train_llm.json`

Add **two** new sections to your `reward_spec.ground_truth`:

- `final_reference`: the **reference final answer** your generator produces *by actually running the plan over MCP tools and analyzing the outputs*. Store both **text** and **facts** (structured), plus optional citations.
- `judge_rubric`: a **structured rubric** your environment will pass to LAJ to score the policy's final text against the reference.

1.1 Minimal schema extension (drop-in)

```json
{
  "data_source": "synthetic/llm",
  "env_class": "MCPToolEnv",
  "prompt": [ { "role": "system", "content": "…" }, { "role": "user", "content": "…" } ],
  "reward_spec": {
    "method": "rule",
    "ground_truth": {
      "task_id": "string",
      "complexity": "simple|moderate|complex",
      "max_turns": 10,
      "limits": { "max_servers": 3, "max_tools": 10 },
      "tool_sequence": [ { "step": 1, "server": "…", "tool": "…", "params": { } }, "…" ],
      "analysis_rubric": {
        "steps": [ { "step": 1, "extract": [], "compute": [], "select": [], "accept_if":
        "final_answer_requirements": {
          "format": "text|markdown|json",
          "must_include": ["list","of","keys-or-names"],
          "grounded_from": ["state_keys_to_check"],
          "quality_criteria": ["no hallucinations","concise", "…"]
        }
      },
      "final_reference": {
        "answer_text": "the reference final summary, generated by your agent after running
        "facts": { "top3": ["NVDA","AMD","META"], "neg_titles": ["…","…"] },
        "citations": { "top3": [3], "neg_titles": [8,9] }  // step indexes that support fa
      },
      "judge_rubric": {
        "weights": { "coverage": 0.3, "grounding": 0.4, "clarity": 0.2, "safety": 0.1 },
        "schema": {
          "type": "object",
          "properties": {
            "coverage": {"type":"number","minimum":0,"maximum":1},
            "grounding":{"type":"number","minimum":0,"maximum":1},
            "clarity":  {"type":"number","minimum":0,"maximum":1},
            "safety":   {"type":"number","minimum":0,"maximum":1},
            "total":    {"type":"number","minimum":0,"maximum":1}
          },
          "required": ["coverage","grounding","clarity","safety","total"]
        },
```

```
        "target_length_range": [40, 140]
      }
    }
  },
  "extra_info": { "scenario": {"scenario":"…","turns": 10} }
}
```

- Keep the **prompt compact** (system + first user) per SkyRL's Dataset Preparation guidance. The env drives multi-turn reward logic. (skyrl.readthedocs.io)
- `final_reference` is **not** shown to the policy; it's used only for reward.

---

2) Changes to your generator `src/dataset/llm/generate_with_llm.py`

You'll add **two phases** after planning:

1. **Execute the plan** over MCP tools, step-by-step, applying your **analysis_rubric.steps** (extract/compute/select/accept_if/next_args_from) to build a **named state** and capture **per-step result summaries**.
2. **Compose the reference final text** using the named state (either via deterministic templating or by calling an LLM with a short, grounded prompt), and write it as `final_reference.answer_text` (+ `facts`, `citations`).

> Tip: keep the **DSL** small and safe (no `eval`), e.g., functions like `last`, `prev`, `pct_change_last_day`, `topk`, `argmax`, `unique`, `regex_extract_all`, `concat`, `head`. This mirrors SkyRL's practice of keeping env logic deterministic and light. (skyrl.readthedocs.io)

2.1 Minimal, PR-ready patch (conceptual diff)

```diff
--- a/src/dataset/llm/generate_with_llm.py
+++ b/src/dataset/llm/generate_with_llm.py
@@
-TASK_SCHEMA = {...}  # your existing schema
+TASK_SCHEMA = {
+  "name": "rl_task",
+  "schema": {
+    "type": "object",
+    "properties": {
+      "task_id": {"type": "string"},
+      "complexity": {"type": "string", "enum": ["simple","moderate","complex"]},
+      "user_prompt": {"type": "string"},
+      "max_turns": {"type": "integer", "minimum": 3, "maximum": 16},
+      "tools_available": {"type": "array", "items": {"type": "string"}},
+      "limits": {"type": "object"},
+      "tool_sequence": {
+        "type": "array", "minItems": 2, "maxItems": 12,
+        "items": {
+          "type": "object",
+          "properties": {
+            "step": {"type":"integer","minimum":1},
+            "server":{"type":"string"},
+            "tool":{"type":"string"},
+            "params":{"type":"object"},
+            "analysis_requirements": {
+              "type":"object",
+              "properties":{
+                "extract":{"type":"array","items":{"type":"string"}},
+                "compute":{"type":"array","items":{"type":"string"}},
+                "select":{"type":"array","items":{"type":"string"}},
+                "accept_if":{"type":"array","items":{"type":"string"}},
+                "next_args_from":{"type":"string"}
+              },
+              "required":["next_args_from"],
+              "additionalProperties": true
+            }
+          },
+          "required": ["step","server","tool","params","analysis_requirements"],
+          "additionalProperties": true
```

```diff
+         }
+       },
+       "final_answer_requirements": {
+         "type": "object",
+         "properties": {
+           "format": {"type":"string"},
+           "must_include":{"type":"array","items":{"type":"string"}},
+           "grounded_from":{"type":"array","items":{"type":"string"}},
+           "quality_criteria":{"type":"array","items":{"type":"string"}}
+         },
+         "required": ["format","must_include","grounded_from"]
+       },
+       "judge_rubric": {
+         "type":"object",
+         "properties":{
+           "weights":{"type":"object"},
+           "schema":{"type":"object"},
+           "target_length_range":{"type":"array","items":{"type":"integer"}}
+         },
+           "required":["weights","schema"]
+       }
+     },
+     "required": ["task_id","complexity","user_prompt","max_turns","tool_sequence","final_a
+     "additionalProperties": false
+   }
+}
@@
 async def _one_task(...):
     # 1) Ask LLM for plan (as you do today)
     task = await _call_llm_with_schema(...)
-    return task
+    # 2) Verify & repair chaining (next_args_from, step ranges)
+    task = _verify_and_repair(task)
+    # 3) Execute plan over MCP tools to build named state and per-step summaries
+    exec_out = await simulate_plan_and_collect(task, tool_manager)
+    # 4) Compose reference final answer (LLM or template), grounded in exec_out.state
+    final_ref = await compose_reference_answer(task, exec_out)
+    task["_exec_out"] = exec_out.to_dict()    # optional: keep light summaries, hashes
+    task["_final_reference"] = final_ref
+    return task
```

```
@@
-def to_skyrl_sample(task, system_prompt):
-    ground_truth = {...}
+def to_skyrl_sample(task, system_prompt):
+    ground_truth = {
+      "task_id": task["task_id"],
+      "complexity": task["complexity"],
+      "max_turns": task["max_turns"],
+      "limits": task.get("limits", {}),
+      "tool_sequence": task["tool_sequence"],
+      "analysis_rubric": {
+        "steps": [{ "step": s["step"], **s["analysis_requirements"] } for s in task["tool
+        "final_answer_requirements": task["final_answer_requirements"]
+      },
+      "final_reference": task["_final_reference"],
+      "judge_rubric": task["judge_rubric"]
+    }
     return {
       "data_source": "synthetic/llm",
       "env_class": "MCPToolEnv",
       "prompt": [
         {"role": "system", "content": system_prompt},
         {"role": "user", "content": task["user_prompt"]}
       ],
-      "reward_spec": {"method":"rule","ground_truth": ground_truth},
+      "reward_spec": {"method":"rule","ground_truth": ground_truth},
       "extra_info": {"version": "lh-v2"}
     }
```

2.2 Execution harness (new helpers)

```
async def simulate_plan_and_collect(task: dict, tm) -> "ExecOut":
    """
    Runs tool_sequence with placeholder resolution and applies analysis_rubric.steps
    to produce named state and lightweight per-step summaries.
    """
    plan = task["tool_sequence"]
    steps_rubric = [s["analysis_requirements"] for s in plan]
    state = {}  # named values introduced by extract/compute/select
    per_step = []
    for idx, step in enumerate(plan, 1):
        args = resolve_placeholders(step["params"], state)   # ${var} replacement
        result = await tm.execute_tool(f"{step['server']}.{step['tool']}", args, timeout=
        summary = summarize_tool_result(result)              # small, e.g., top keys, co
        # apply rubric: extract/compute/select/accept_if
        updates, checks = apply_analysis(steps_rubric[idx-1], result, state)
        state.update(updates)
        per_step.append({"step": idx, "args": args, "summary": summary, "checks": checks}
    return ExecOut(state=state, steps=per_step)

def apply_analysis(ar: dict, result: dict, state: dict) -> tuple[dict, dict]:
    """Implements a tiny, safe DSL (no eval) for extract/compute/select/accept_if."""
    updates = {}
    checks = {"accept_pass": True, "missing": []}
    # extract
    for name in ar.get("extract", []):
        val, ok = safe_extract(name, result)   # support "close[]", "articles[][title]"
        if ok: updates[name.split("[")[0]] = val
        else:  checks["missing"].append(name); checks["accept_pass"] = False
    # compute/select (whitelisted ops like last, prev, pct_change_last_day, topk, argmax,
    for expr in ar.get("compute", []): updates.update(safe_compute(expr, state))
    for expr in ar.get("select", []):  updates.update(safe_compute(expr, state))
    # accept_if
    for cond in ar.get("accept_if", []):
        if not safe_check(cond, state): checks["accept_pass"] = False
    return updates, checks

async def compose_reference_answer(task: dict, exec_out: "ExecOut") -> dict:
    """Produce final_reference {answer_text, facts, citations} from state."""
    far = task["final_answer_requirements"]
```

```
    # build facts deterministically from state:
    facts = build_facts(exec_out.state, far.get("grounded_from", []))
    # use a small LLM prompt or a template to compose the text
    answer_text = await small_llm_compose(far, exec_out.state, facts)
    # map which steps support which facts
    citations = infer_citations(facts, exec_out.steps)
    return {"answer_text": answer_text, "facts": facts, "citations": citations}
```

- **Why do this at data-gen time?** So your **reference** is grounded in **real tool results**, not a hallucinated answer. At training, the env will compare the **policy's** final text to this reference using LAJ and/or heuristics.

---

3) Environment changes (how the final answer is rewarded)

Your `MCPToolEnv(BaseTextEnv)` should accept either:

- a **tool call** (`{"tool":"server.tool","arguments":{...}}`) or
- a **final answer** (`{"final_answer":"…text…"}`), similar to the **Search example**, which uses `<answer>…</answer>` with stop strings to mark the last turn. (skyrl.readthedocs.io)

3.1 Final answer scoring in the env

Heuristic (fast):

- **Coverage:** does text include all names in `analysis_rubric.final_answer_requirements.must_include` ?
- **Grounding:** is text consistent with `final_reference.facts` (e.g., set overlap equals 1.0), and **does not** contradict them?
- **Clarity:** target length range; simple readability checks.
- **Safety:** blacklist checks.

LLM-as-a-Judge (LAJ):

- Judge sees a **compact state summary**, the **policy's final text**, and the **reference** (text +

facts).
- Use **Structured Outputs** with your `judge_rubric.schema` to force **numbers-only**.

Reward:

- (r_{\text{final}} = \lambda_{\text{heur}} r_{\text{heur}} + \lambda_{\text{laj}} r_{\text{laj}}) (both in ([0,1])).
- Total episode return is sum of shaped tool rewards + (\alpha \cdot r_{\text{final}}).

This mirrors SkyRL's pattern where the final text inside `<answer>` is what gets rewarded (binary in Search; graded here). (skyrl.readthedocs.io)

---

4) Two sample dataset items (with `final_reference` & `judge_rubric`)

4.1 NASDAQ-100 news triage (truncated for space; drop into your JSON array)

```
{
  "data_source": "synthetic/llm",
  "env_class": "MCPToolEnv",
  "prompt": [
    { "role": "system", "content": "You have tools DuckDuckGo, yahoo_finance, python_execu
    { "role": "user", "content": "Find top-3 gainers in NASDAQ-100 today, get 5 news head
  ],
  "reward_spec": {
    "method": "rule",
    "ground_truth": {
      "task_id": "nasdaq100_neg_digest_v2",
      "complexity": "complex",
      "max_turns": 12,
      "limits": { "max_servers": 3, "max_tools": 10 },
      "tool_sequence": [
        { "step": 1, "server": "DuckDuckGo", "tool": "search", "params": { "query": "NASD
        { "step": 2, "server": "DuckDuckGo", "tool": "fetch_content", "params": { "url": "
        { "step": 3, "server": "yahoo_finance", "tool": "get_yfinance_price_history", "param
        { "step": 4, "server": "python_execution", "tool": "python_execution", "params": {
        { "step": 5, "server": "DuckDuckGo", "tool": "search", "params": { "query": "${top
        { "step": 6, "server": "DuckDuckGo", "tool": "search", "params": { "query": "${top
        { "step": 7, "server": "DuckDuckGo", "tool": "search", "params": { "query": "${top
        { "step": 8, "server": "python_execution", "tool": "python_execution", "params": {
        { "step": 9, "server": "python_execution", "tool": "python_execution", "params": {
        { "step": 10, "server": "slack", "tool": "send_slack_message", "params": { "channe
      ],
      "analysis_rubric": {
        "steps": [
          { "step": 1, "extract": ["tickers_url"], "compute": [], "select": [], "accept_i
          { "step": 2, "extract": ["content"], "compute": ["tickers = regex_extract_all('
          { "step": 3, "extract": ["price_json"], "compute": ["pct = pct_change_last_day(p
          { "step": 4, "extract": ["top3[]"], "compute": [], "select": [], "accept_if": ["
          { "step": 5, "extract": ["articles0[][title]"], "compute": ["news_titles0 = titl
          { "step": 6, "extract": ["articles1[][title]"], "compute": ["news_titles1 = titl
          { "step": 7, "extract": ["articles2[][title]"], "compute": ["news_titles2 = titl
          { "step": 8, "extract": ["title_sentiment_map{title->score}"], "compute": ["news
          { "step": 9, "extract": ["neg_titles[]"], "compute": [], "select": ["neg_titles
          { "step": 10, "extract": [], "compute": [], "select": [], "accept_if": [], "nex
        ],
```

```
        "final_answer_requirements": {
          "format": "markdown",
          "must_include": ["top3", "neg_titles"],
          "grounded_from": ["top3", "title_sentiment_map"],
          "quality_criteria": ["relevant headlines", "no hallucinated tickers", "concise"]
        }
      },
      "final_reference": {
        "answer_text": "Top-3 NASDAQ-100 gainers today: NVDA, AMD, META. Notable negative
        "facts": { "top3": ["NVDA","AMD","META"], "neg_titles": ["…","…","…"] },
        "citations": { "top3": [4], "neg_titles": [8,9] }
      },
      "judge_rubric": {
        "weights": { "coverage": 0.35, "grounding": 0.4, "clarity": 0.15, "safety": 0.10 }
        "schema": {
          "type": "object",
          "properties": {
            "coverage": {"type":"number","minimum":0,"maximum":1},
            "grounding":{"type":"number","minimum":0,"maximum":1},
            "clarity":  {"type":"number","minimum":0,"maximum":1},
            "safety":   {"type":"number","minimum":0,"maximum":1},
            "total":    {"type":"number","minimum":0,"maximum":1}
          },
          "required": ["coverage","grounding","clarity","safety","total"]
        },
        "target_length_range": [40, 140]
      }
    }
  }
}
```

4.2 S3 error histogram (final answer includes decision & justification)

```
{
  "data_source": "synthetic/llm",
  "env_class": "MCPToolEnv",
  "prompt": [
    { "role": "system", "content": "You have tools aws, python_execution, slack, jira. Emi
    { "role": "user", "content": "Find buckets >10 GB, compute today's error %, post a his
  ],
  "reward_spec": {
    "method": "rule",
    "ground_truth": {
      "task_id": "aws_error_histogram_v2",
      "complexity": "complex",
      "max_turns": 12,
      "limits": { "max_servers": 4, "max_tools": 10 },
      "tool_sequence": [
        { "step": 1, "server": "aws", "tool": "aws_s3_list_buckets", "params": {} },
        { "step": 2, "server": "python_execution", "tool": "python_execution", "params": {
        { "step": 3, "server": "aws", "tool": "aws_s3_list_objects", "params": { "bucket"
        { "step": 4, "server": "python_execution", "tool": "python_execution", "params": {
        { "step": 5, "server": "aws", "tool": "aws_s3_list_objects", "params": { "bucket"
        { "step": 6, "server": "python_execution", "tool": "python_execution", "params": {
        { "step": 7, "server": "python_execution", "tool": "python_execution", "params": {
        { "step": 8, "server": "slack", "tool": "send_slack_message", "params": { "channel
        { "step": 9, "server": "python_execution", "tool": "python_execution", "params": {
        { "step": 10, "server": "jira", "tool": "create_ticket", "params": { "project": "
        { "step": 11, "server": "slack", "tool": "send_slack_message", "params": { "channe
      ],
      "analysis_rubric": {
        "steps": [
          { "step": 1, "extract": ["buckets[]"], "compute": [], "select": [], "accept_if"
          { "step": 2, "extract": ["big_buckets[]"], "compute": [], "select": [], "accept_
          { "step": 3, "extract": ["objects_json"], "compute": [], "select": [], "accept_i
          { "step": 4, "extract": ["error_pct1"], "compute": [], "select": [], "accept_if"
          { "step": 5, "extract": ["objects_json2"], "compute": [], "select": [], "accept_
          { "step": 6, "extract": ["error_pct2"], "compute": ["error_pct_map = merge_map(
          { "step": 7, "extract": ["histogram_path"], "compute": [], "select": [], "accep
          { "step": 8, "extract": [], "compute": [], "select": [], "accept_if": [], "next_
          { "step": 9, "extract": ["exceed_buckets[]"], "compute": [], "select": [], "acce
          { "step": 10, "extract": ["jira_id"], "compute": [], "select": [], "accept_if":
```

```
        { "step": 11, "extract": [], "compute": [], "select": [], "accept_if": [], "nex
      ],
      "final_answer_requirements": {
        "format": "markdown",
        "must_include": ["histogram_path", "exceed_buckets"],
        "grounded_from": ["error_pct_map","exceed_buckets","jira_id"],
        "quality_criteria": ["include only buckets above threshold", "valid Jira ID if
      }
    },
    "final_reference": {
      "answer_text": "Histogram posted. Buckets above 1%: logs-prod, archive-east. Jira
      "facts": { "exceed_buckets": ["logs-prod","archive-east"], "jira_id": "OPS-1432",
      "citations": { "exceed_buckets": [9], "jira_id": [10], "histogram_path": [7] }
    },
    "judge_rubric": {
      "weights": { "coverage": 0.3, "grounding": 0.4, "clarity": 0.2, "safety": 0.1 },
      "schema": {
        "type": "object",
        "properties": {
          "coverage": {"type":"number","minimum":0,"maximum":1},
          "grounding":{"type":"number","minimum":0,"maximum":1},
          "clarity":  {"type":"number","minimum":0,"maximum":1},
          "safety":   {"type":"number","minimum":0,"maximum":1},
          "total":    {"type":"number","minimum":0,"maximum":1}
        },
        "required": ["coverage","grounding","clarity","safety","total"]
      },
      "target_length_range": [40, 160]
    }
  }
 }
}
```

---

5) How this plugs into SkyRL

- **Dataset**: compact seed prompt + rich `ground_truth` metadata — exactly how SkyRL expects you to provide data that the env will use for rewards. (skyrl.readthedocs.io)

- **Environment**: subclass `BaseTextEnv`, parse tool calls & final answer, call your ToolGroups, compute rewards per turn. This is the documented pattern for multi-turn tasks with tools and final text (see **Search** example: “agent has n turns, outputs `<answer>` at the end, receive reward”). (skyrl.readthedocs.io)
- **Training**: run GRPO in SkyRL-Train exactly like their examples; the final answer reward is just another term. (If you’re using vLLM for rollout, ensure you capture per-token logprobs and refresh LoRA adapters after updates.)

---

6) Gotchas & best practices

- **Leakage:** never put `final_reference` in observations; it’s only for reward.
- **Cost:** cache LAJ calls by `(task_id, hash(policy_answer))`.
- **Stability:** start with heavier heuristic weight (e.g., 0.7) and add LAJ (0.3).
- **Style mismatch:** LAJ should judge **facts & grounding** more than surface style; include `facts` in the rubric context.
- **Determinism:** keep DSL deterministic; if a step fails `accept_if`, penalize and keep going to preserve learning signal.
- **Search-style tags (optional):** if you adopt `<answer>…</answer>`, add stop strings so sampler cuts generations cleanly (the Search example does this with `</answer>`). (skyrl.readthedocs.io)

If you paste your current `generate_with_llm.py`, I’ll return a **line-exact patch** adding:

- the extended schema,
- the MCP execution harness,
- the DSL evaluator,
- and the `final_reference` + `judge_rubric` writing logic.

That will give you a **self-contained** synthetic dataset where **plans are executed, analysis is applied, and reference answers are produced**, ready for **LAJ-based** final scoring during SkyRL training.

Implementation plan for the data generator and environment design

Below is a **surgical, code-level plan** to (A) upgrade your `src/dataset/llm/generate_with_llm.py` so the synthetic agent **actually executes its tool plan** and emits a **ground-truth final answer**, and (B) implement a **multi-turn SkyRL environment** that rewards (1) correct tool use + analysis each turn **and** (2) the **final text answer** using **LLM-as-a-Judge (LAJ)** (optionally combined with heuristics).

Where I reference SkyRL behavior, I cite the official docs/examples so your team can confirm alignment:

- **Dataset format** (what each record must contain). (skyrl.readthedocs.io)
- **BaseTextEnv** interface and how multi-turn envs are implemented. (skyrl.readthedocs.io)
- **Tools integration** (ToolGroup, parsing actions, calling tools). (skyrl.readthedocs.io)
- **Multi-turn final answers** (Search example uses `<answer>…</answer>` and stop strings). (skyrl.readthedocs.io)
- **LLM-as-a-Judge** (reference example + config pattern). (skyrl.readthedocs.io)

> I reviewed the three files you attached: `generate_with_llm.py`, `mini_agent_trajectories.py`, and `common.py`. They appear abbreviated with ellipses (`...`). I'll give **drop-in code blocks** and **exact insertion points** so you can merge them even if lines don't match 1:1.

A) Upgrade `generate_with_llm.py` to produce *executed*, *grounded* final answers

Goal: after your LLM proposes a multi-turn tool plan, **actually run** that plan against your MCP servers (or the same local tool shims you'll use in the env), apply an **analysis DSL** (extract/compute/select/accept_if), then **compose a reference final answer** from the derived state. The resulting dataset entries add:

- `ground_truth.tool_sequence` (what to do),

- `ground_truth.analysis_rubric` (how to check each step),
- **`ground_truth.final_reference`** (answer text + facts + citations),
- **`ground_truth.judge_rubric`** (weights + JSON schema the env passes to the Judge).

SkyRL's dataset loader wants **compact prompts** + **reward_spec** per sample; the rest is your metadata for the environment to compute rewards. (skyrl.readthedocs.io)

A.1 Minimal schema extension

Add two blocks to your task schema + final writer:

- `final_answer_requirements` (format, must_include, grounded_from, criteria)
- `judge_rubric` (weights + structured output schema + optional length range)

> These are used for **data generation** (to force the planner to think about the end state) **and** later by the **environment** to score final answers.

🔧 Patch: add/replace these pieces in `src/dataset/llm/generate_with_llm.py`

> ### 1) Imports & helpers (top of file, after existing imports)

```
# === NEW: imports for executing plan & composing final ===
from dataclasses import dataclass
from copy import deepcopy

# If you already have a ToolManager for MCP in your repo, import it here.
# Otherwise, you can stub `execute_tool_fqn(tool_fqn: str, params: dict) -> dict`
# and later wire it to your MCP client used by the environment.
try:
    from src.utils.tool_manager import ToolManager
    MCP_AVAILABLE = True
except Exception:
    MCP_AVAILABLE = False
    ToolManager = None
```

> ### 2) Extend your TASK_SCHEMA (replace/augment your dict)

```
TASK_SCHEMA = {
    "name": "skyrl_task",
    "schema": {
        "type": "object",
        "properties": {
            "task_id": {"type": "string"},
            "user_prompt": {"type": "string"},
            "complexity": {"type": "string", "enum": ["simple", "moderate", "complex"]},
            "max_turns": {"type": "integer", "minimum": 2, "maximum": 20},
            "tools_available": {"type": "array", "items": {"type": "string"}},
            "limits": {"type": "object"},
            "tool_sequence": {
                "type": "array",
                "minItems": 2,
                "maxItems": 16,
                "items": {
                    "type": "object",
                    "properties": {
                        "step": {"type": "integer", "minimum": 1},
                        "server": {"type": "string"},
                        "tool": {"type": "string"},
                        "params": {"type": "object"},
                        "analysis_requirements": {
                            "type": "object",
                            "properties": {
                                "extract": {"type": "array", "items": {"type":"string"}},
                                "compute": {"type": "array", "items": {"type":"string"}},
                                "select":  {"type": "array", "items": {"type":"string"}},
                                "accept_if": {"type":"array", "items":{"type":"string"}},
                                "next_args_from": {"type":"string"}
                            },
                            "required": ["next_args_from"]
                        }
                    },
                    "required": ["step", "server", "tool", "params", "analysis_requiremen
                    "additionalProperties": True
                }
            },
            "final_answer_requirements": {
```

```
                "type": "object",
                "properties": {
                    "format": {"type": "string"},  # "text" | "markdown" | "json"
                    "must_include": {"type": "array", "items": {"type":"string"}},
                    "grounded_from": {"type": "array", "items": {"type":"string"}},
                    "quality_criteria": {"type": "array", "items": {"type":"string"}}
                },
                "required": ["format", "must_include", "grounded_from"]
            },
            "judge_rubric": {
                "type": "object",
                "properties": {
                    "weights": {"type": "object"},
                    "schema": {"type": "object"},             # JSON schema for LAJ struc
                    "target_length_range": {"type": "array", "items": {"type":"integer"},
                },
                "required": ["weights", "schema"]
            }
        },
        "required": ["task_id", "user_prompt", "complexity", "max_turns", "tool_sequence",
                     "final_answer_requirements", "judge_rubric"],
        "additionalProperties": True
    }
}
```

3) Add an execution record and DSL to evaluate steps (near bottom or a new section)

```python
# === NEW: Executed plan outputs ===
@dataclass
class ExecStep:
    step: int
    tool_fqn: str
    args: dict
    result_summary: dict
    accept_pass: bool
    checks: dict

@dataclass
class ExecOut:
    state: dict
    steps: List[ExecStep]

# --- Safe DSL utilities (expand as needed; keep deterministic) ---
def _extract_path(result: Any, path: str) -> Tuple[Optional[Any], bool]:
    """
    Supports simple paths like 'field', 'field[]', 'obj[][title]', 'map{key->val}' summary
    Return (value, ok)
    """
    try:
        if path.endswith("[]"):  # list extraction
            key = path[:-2]
            return result.get(key, []), True
        if path.endswith("[][title]"):
            key = path.split("[]")[0]
            items = result.get(key, [])
            return [it.get("title") for it in items if isinstance(it, dict)], True
        if "{title->score}" in path:
            # summarized maps from list of items having title/score
            base = path.split("{")[0]
            items = result.get(base, [])
            return {it["title"]: it.get("score", 0.0) for it in items if "title" in it},
        return result.get(path, None), (path in result)
    except Exception:
        return None, False

def _compute(expr: str, state: dict) -> dict:
```

```python
    """
    Tiny, whitelisted DSL. Examples:
    - "pct = pct_change_last_day(price_json)"
    - "top3 = topk(pct, 3)"
    - "tickers = regex_extract_all('[A-Z]{1,5}', content)"
    - "neg_titles = head(neg_titles, 5)"
    """
    out = {}
    name, rhs = [s.strip() for s in expr.split("=", 1)]
    def pct_change_last_day(price_json):
        # price_json: {ticker: [ {open,close,...}, ... ]}
        # return {ticker: pct} for last 2 rows
        pct = {}
        for k, arr in price_json.items():
            if len(arr) >= 2 and "close" in arr[-1] and "close" in arr[-2]:
                b, a = float(arr[-2]["close"]), float(arr[-1]["close"])
                if b != 0:
                    pct[k] = (a / b) - 1.0
        return pct
    def topk(d: dict, k: int):
        return [k_ for k_, _ in sorted(d.items(), key=lambda kv: kv[1], reverse=True)[:k]]
    def head(lst: list, n: int): return lst[:n]
    def unique(lst: list): return list(dict.fromkeys(lst))
    def concat(*lsts): out=[]; [out.extend(_l) for _l in lsts]; return out
    def count_keys(d: dict): return len(d.keys()) if isinstance(d, dict) else 0
    def regex_extract_all(pattern: str, text: str):
        import re
        return re.findall(pattern, text or "")
    # ---- Eval RHS in controlled namespace ----
    safe_ns = {
        **deepcopy(state),
        "pct_change_last_day": pct_change_last_day,
        "topk": topk, "head": head, "unique": unique, "concat": concat,
        "count_keys": count_keys, "regex_extract_all": regex_extract_all,
    }
    # Support calls in the form fn(state_key, ...), where state_key was introduced earlie
    value = eval(rhs, {"__builtins__": {}}, safe_ns)  # guarded namespace
    out[name] = value
    return out
```

```python
def _check(cond: str, state: dict) -> bool:
    # e.g., "len(tickers) >= 80", "top_gainer in tickers", "histogram_path ~= '^/tmp/.*\\
    try:
        if " ~=" in cond:
            lhs, pattern = [s.strip() for s in cond.split("~=", 1)]
            import re
            return re.search(pattern.strip("'\""), str(eval(lhs, {"__builtins__": {}}, sta
        return bool(eval(cond, {"__builtins__": {}}, state))
    except Exception:
        return False

def _resolve_placeholders(obj: Any, state: dict) -> Any:
    # Replace ${var} or ${arr[i]} inside params
    if isinstance(obj, str):
        import re
        def repl(m):
            key = m.group(1)
            try:
                return str(eval(key, {"__builtins__": {}}, state))
            except Exception:
                return m.group(0)
        return re.sub(r"\$\{([^}]+)\}", repl, obj)
    if isinstance(obj, dict):
        return {k: _resolve_placeholders(v, state) for k, v in obj.items()}
    if isinstance(obj, list):
        return [_resolve_placeholders(v, state) for v in obj]
    return obj
```

4) Execute the plan + build a final reference

```python
async def simulate_plan_and_collect(task: dict, tm: Optional[ToolManager]) -> ExecOut:
    """
    Runs tool_sequence with placeholder resolution and applies analysis_requirements
    to produce named state + per-step summaries.
    """
    state: dict = {}
    exec_steps: List[ExecStep] = []
    for step_obj in task["tool_sequence"]:
        step = int(step_obj["step"])
        tool_fqn = f'{step_obj["server"]}.{step_obj["tool"]}'
        params = _resolve_placeholders(step_obj.get("params", {}), state)

        if tm is None:
            # If MCP ToolManager not wired yet, just mock a stable shape
            result = {"ok": True, "echo": params}
        else:
            result = await tm.execute_tool(tool_fqn, params, timeout=20.0)

        # Apply analysis requirements
        ar = step_obj.get("analysis_requirements", {})
        updates = {}
        missing = []
        accept = True
        for need in ar.get("extract", []):
            val, ok = _extract_path(result, need)
            if ok:
                key = need.split("[")[0].split("{")[0]
                updates[key] = val
            else:
                missing.append(need)
                accept = False
        for expr in ar.get("compute", []):
            try: updates.update(_compute(expr, {**state, **updates}))
            except Exception: accept = False
        for expr in ar.get("select", []):
            try: updates.update(_compute(expr, {**state, **updates}))
            except Exception: accept = False
        for cond in ar.get("accept_if", []):
            if not _check(cond, {**state, **updates}):
```

```
                accept = False

        state.update(updates)
        exec_steps.append(ExecStep(
            step=step, tool_fqn=tool_fqn, args=params,
            result_summary={"keys": list(result)[:10]},  # keep tiny
            accept_pass=accept,
            checks={"missing": missing, "updated": list(updates.keys())}
        ))
    return ExecOut(state=state, steps=exec_steps)

async def compose_reference_answer(task: dict, exec_out: ExecOut, client: AsyncOpenAI) ->
    far = task["final_answer_requirements"]
    facts = {name: exec_out.state.get(name) for name in far.get("grounded_from", [])}
    # Compose with a small LLM, STRICT JSON output to reduce drift
    system = "You are a concise analyst. Write the final answer strictly grounded in the p
    user = {
        "facts": facts,
        "must_include": far.get("must_include", []),
        "format": far.get("format", "text"),
        "quality_criteria": far.get("quality_criteria", []),
    }
    # If you prefer determinism or cost-free, you can template instead of calling the LLM
    resp = await client.chat.completions.create(
        model=os.getenv("OPENAI_RESP_MODEL", "gpt-4o-mini"),
        response_format={"type": "json_object"},
        messages=[
            {"role": "system", "content": system},
            {"role": "user", "content": json.dumps(user)}
        ],
        temperature=0.1,
    )
    data = json.loads(resp.choices[0].message.content)
    answer_text = data.get("answer") or data.get("final_answer") or json.dumps(facts)
    # Build simple citations: map each fact name to the last step that updated it
    name_to_step = {}
    for s in reversed(exec_out.steps):
        for k in exec_out.state.keys():
            if k not in name_to_step and k in s.checks.get("updated", []):
                name_to_step[k] = s.step
```

```
    citations = {k: [name_to_step.get(k)] for k in facts.keys() if name_to_step.get(k)}
    return {"answer_text": answer_text, "facts": facts, "citations": citations}
```

> **5) Thread it into your existing `_one_task(...)` flow** (after you get the *planned* `task` from the LLM)

```
async def _one_task(...):
    task = await _call_llm_with_schema(..., TASK_SCHEMA, ...)   # your existing planner ca
    task = _verify_and_repair(task)  # ensure next_args_from chains exist (implement simpl

    tm = ToolManager() if MCP_AVAILABLE else None  # wire to MCP servers if available
    exec_out = await simulate_plan_and_collect(task, tm)

    openai_client = AsyncOpenAI(api_key=os.environ["OPENAI_API_KEY"])
    final_ref = await compose_reference_answer(task, exec_out, openai_client)

    task["_exec_out"] = {"steps": [asdict(s) for s in exec_out.steps]}  # optional, keep t
    task["_final_reference"] = final_ref
    return task
```

> **6) Write the SkyRL-compatible record** (modify your writer)

```python
def to_skyrl_sample(task: dict, system_prompt: str, env_class: str, data_source: str) ->
    prompt_messages = [
        {"role": "system", "content": system_prompt},
        {"role": "user",   "content": task["user_prompt"]},
    ]
    ground_truth = {
        "task_id": task["task_id"],
        "complexity": task["complexity"],
        "max_turns": task["max_turns"],
        "limits": task.get("limits", {}),
        "tool_sequence": task["tool_sequence"],
        "analysis_rubric": {
            "steps": [
                {"step": s["step"], **s["analysis_requirements"]}
                for s in task["tool_sequence"]
            ],
            "final_answer_requirements": task["final_answer_requirements"]
        },
        "final_reference": task["_final_reference"],
        "judge_rubric": task["judge_rubric"],
    }
    return {
        "data_source": data_source,
        "env_class": env_class,
        "prompt": prompt_messages,
        "reward_spec": {"method": "rule", "ground_truth": ground_truth},
        "extra_info": {"gen": {"model": task.get("_model"), "backend": task.get("_backend"
    }
```

That's all you need on the data-gen side. Each sample is now **grounded**: the plan was **executed**, the **state** was derived by your DSL, and a **reference final answer** exists for the **Judge** to compare against during training. (Dataset remains within the SkyRL format: minimal prompt + `reward_spec` — everything else lives under ground_truth for env reward.) (skyrl.readthedocs.io)

B) Implement the multi-turn environment with per-turn rewards + final text reward

Create `src/envs/mcp_tool_env.py` (or extend your existing one) as a subclass of `BaseTextEnv` (text-in/text-out). In each step:

1. **Parse action**:
   - either a **tool call**: `{"tool":"server.tool","arguments":{...}}`
   - or a **final answer**: `{"final_answer":"…text…"}`
     (You can also allow `<answer>…</answer>` or `<tool>…</tool>` tags; if you do, add **stop strings** like the Search example: `stop='["</tool>", "</answer>"]'` in generator sampling.) (skyrl.readthedocs.io)
2. **Tool step**:
   - resolve placeholders from the **current named state**, call the right ToolGroup method, run **analysis rubric**: `extract/compute/select/accept_if/next_args_from`.
   - compute **shaped reward** (+ penalties if malformed).
3. **Final step**:
   - compute **Heuristic** score (coverage/grounding/clarity/safety vs the requirements + reference facts).
   - compute **LAJ** score by sending a compact rubric to the Judge model with **structured outputs** (numbers-only). SkyRL already documents the judge pattern (for GSM8K) — we just do a graded version with multiple dimensions. (skyrl.readthedocs.io)

Here's a concise implementation skeleton:

```python
# src/envs/mcp_tool_env.py
from __future__ import annotations
import json, re, os, asyncio
from typing import Any, Dict, List, Tuple, Optional
from skyrl_gym.envs.base_text_env import BaseTextEnv, BaseTextEnvStepOutput  # API per do
from skyrl_gym.tools.core import ToolGroup  # or your ToolManager wrappers
from openai import AsyncOpenAI

class MCPToolEnv(BaseTextEnv):
    """Multi-turn, multi-tool env that also rewards a final text answer."""
    def __init__(self, env_config: Dict[str, Any] = None, extras: Dict[str, Any] = None):
        super().__init__()
        self.turn = 0
        self.max_turns = (env_config or {}).get("max_turns", 8)
        self.reward_weights = (env_config or {}).get("reward_weights", {
            "tool_name": 0.2, "param_binding": 0.15, "extract": 0.15,
            "compute": 0.15, "accept_if": 0.1, "penalty": -0.1,
            "final_heur": 0.6, "final_laj": 0.4
        })
        # Initialize ToolGroups (search, python, finance, slack, etc.)
        # Or bridge to your MCP servers if you already have a ToolManager.
        self.init_tool_groups(self._make_tool_groups(extras or {}))
        # Judge
        self.judge = AsyncOpenAI(api_key=os.environ.get("OPENAI_API_KEY"))

        # Set by init()
        self.gt = None            # ground_truth (dict from reward_spec)
        self.state = {}           # named values we derive per step

    def _make_tool_groups(self, extras: Dict[str, Any]) -> List[ToolGroup]:
        # TODO: return actual tool groups; you can wrap MCP servers as ToolGroup adapters
        return []

    # ---------- BaseTextEnv API ----------
    def init(self, prompt, ground_truth=None) -> Tuple[list, dict]:
        """
        Called once per episode with the dataset prompt and its ground_truth from reward_
        """
        self.turn = 0
```

```
        self.state = {}
        self.gt = ground_truth or {}
        return prompt, {"task_id": self.gt.get("task_id")}

    def step(self, action: str) -> BaseTextEnvStepOutput:
        self.turn += 1
        kind, payload = self._parse_action(action)
        if kind == "tool":
            tool_fqn, args = payload["name"], payload.get("arguments", {})
            # Execute the tool via BaseTextEnv helper
            try:
                group_name, tool_name = tool_fqn.split(".", 1)
                obs = self._execute_tool(group_name, tool_name, [args])  # returns seriali
            except Exception as e:
                # malformed tool -> penalty, continue
                return BaseTextEnvStepOutput(
                    observations=[{"role": "user", "content": json.dumps({"error": str(e)}
                    reward=self.reward_weights["penalty"],
                    done=(self.turn >= self.max_turns),
                    metadata={"error": "tool_exec", "exception": str(e)}
                )
            # Apply analysis rubric for this step (if present)
            step_idx = self._match_step(tool_fqn)
            r_step, meta = self._score_tool_step(step_idx, tool_fqn, args, obs)
            done = (self.turn >= self.max_turns)
            return BaseTextEnvStepOutput(
                observations=[{"role": "user", "content": json.dumps(obs)[:2048]}],
                reward=r_step,
                done=done,
                metadata={"step": step_idx, **meta}
            )

        # Final answer branch
        final_text = payload
        r_final, meta = asyncio.get_event_loop().run_until_complete(
            self._score_final(final_text)
        )
        return BaseTextEnvStepOutput(
            observations=[],
            reward=r_final,
```

```
            done=True,
            metadata={"final": meta}
        )

    # ---------- Parsing ----------
    def _parse_action(self, text: str) -> Tuple[str, Any]:
        # JSON first
        try:
            obj = json.loads(text)
            if "tool" in obj:
                return "tool", {"name": obj["tool"], "arguments": obj.get("arguments", {}
            if "final_answer" in obj:
                return "final", obj["final_answer"]
        except Exception:
            pass
        # Tag-style, compatible with Search pattern
        if "<answer>" in text and "</answer>" in text:
            ans = text.split("<answer>")[1].split("</answer>")[0].strip()
            return "final", ans
        if "<tool>" in text and "</tool>" in text:
            inner = text.split("<tool>")[1].split("</tool>")[0]
            m = re.search(r"<([a-zA-Z0-9_.-]+)>(.*)</\\1>", inner, re.DOTALL)
            if m:
                name, args_text = m.group(1), m.group(2).strip()
                try:
                    args = json.loads(args_text)
                except Exception:
                    args = {"raw": args_text}
                return "tool", {"name": name, "arguments": args}
        # Default: treat as final free text
        return "final", text.strip()

    # ---------- Rewarding ----------
    def _match_step(self, tool_fqn: str) -> int:
        for s in self.gt.get("tool_sequence", []):
            if f'{s["server"]}.{s["tool"]}' == tool_fqn:
                return int(s["step"])
        return self.turn  # fallback

    def _score_tool_step(self, step_idx: int, tool_fqn: str, args: dict, result: dict) ->
```

```python
        ar = None
        for s in self.gt.get("analysis_rubric", {}).get("steps", []):
            if int(s["step"]) == step_idx:
                ar = s; break
        if ar is None:
            # No rubric -> small neutral reward
            return 0.0, {"warn": "no_rubric"}

        reward = 0.0
        meta = {"tool": tool_fqn, "args": args, "accept_if": []}

        # Tool choice reward
        expected_fqn = None
        for st in self.gt["tool_sequence"]:
            if int(st["step"]) == step_idx:
                expected_fqn = f'{st["server"]}.{st["tool"]}'; break
        if expected_fqn == tool_fqn:
            reward += self.reward_weights["tool_name"]

        # Param binding reward (did args use the prior state's 'next_args_from' value?)
        naf = ar.get("next_args_from")
        if naf:
            used = json.dumps(args)
            if naf in used:
                reward += self.reward_weights["param_binding"]

        # Extract
        ext_ok = True
        for need in ar.get("extract", []):
            val = self._extract_path(result, need)[0]
            if val is None: ext_ok = False
            else:
                self.state[need.split("[")[0].split("{")[0]] = val
        if ext_ok: reward += self.reward_weights["extract"]

        # Compute/select
        try:
            for expr in ar.get("compute", []): self.state.update(self._compute(expr))
            for expr in ar.get("select", []):  self.state.update(self._compute(expr))
            reward += self.reward_weights["compute"]
```

```
        except Exception:
            pass

        # accept_if
        all_ok = True
        for cond in ar.get("accept_if", []):
            ok = self._check(cond)
            meta["accept_if"].append({"cond": cond, "ok": ok})
            all_ok = all_ok and ok
        if all_ok: reward += self.reward_weights["accept_if"]

        return reward, meta

    async def _score_final(self, text: str) -> Tuple[float, dict]:
        # Heuristic
        h_score, h_meta = self._score_final_heur(text)
        # LAJ
        j_score, j_meta = await self._score_final_laj(text)
        w = self.reward_weights
        total = w["final_heur"] * h_score + w["final_laj"] * j_score
        return float(total), {"heur": h_meta, "laj": j_meta}

    def _score_final_heur(self, text: str) -> Tuple[float, dict]:
        far = self.gt["analysis_rubric"]["final_answer_requirements"]
        jr  = self.gt["judge_rubric"]
        weights = jr["weights"]
        lo, hi = (jr.get("target_length_range") or [0, 10**9])

        # Coverage: all must_include items appear or can be verified from state
        must = far.get("must_include", [])
        cov_hits = sum(1 for k in must if k in text or k in json.dumps(self.state))
        coverage = cov_hits / max(1, len(must))

        # Grounding: mentions consistent with reference facts
        facts = self.gt["final_reference"]["facts"]
        grounding = self._grounding_score(text, facts)

        # Clarity: simple length band
        words = len(text.split())
        clarity = 1.0 if lo <= words <= hi else 0.5 if (0.7*lo) <= words <= (1.5*hi) else
```

```python
        # Safety: simple heuristic
        safety = 1.0 if not re.search(r"\b(SSN|password|api_key)\b", text, re.I) else 0.0

        total = (weights.get("coverage",0)*coverage +
                 weights.get("grounding",0)*grounding +
                 weights.get("clarity",0)*clarity +
                 weights.get("safety",0)*safety)
        return float(total), {"coverage":coverage, "grounding":grounding, "clarity":clarity

    async def _score_final_laj(self, text: str) -> Tuple[float, dict]:
        # Compact state summary + reference
        facts = self.gt["final_reference"]["facts"]
        ref   = self.gt["final_reference"]["answer_text"]
        schema = self.gt["judge_rubric"]["schema"]
        rubric_prompt = {
            "instructions": [
                "Score the FINAL answer vs the reference & facts.",
                "Return ONLY JSON with fields in the provided schema."
            ],
            "facts": facts, "reference": ref, "final": text
        }
        resp = await self.judge.chat.completions.create(
            model=os.getenv("OPENAI_JUDGE_MODEL", "gpt-4o-mini"),
            response_format={"type": "json_schema", "json_schema": {"name": "judge_schema"
            messages=[
                {"role":"system","content":"You are a strict evaluator."},
                {"role":"user","content": json.dumps(rubric_prompt)}
            ],
            temperature=0.0,
        )
        data = json.loads(resp.choices[0].message.content)
        return float(data.get("total", 0.0)), data

    # --- local copies of DSL helpers (same semantics as generator) ---
    def _extract_path(self, res: dict, path: str): return _extract_path(res, path)
    def _compute(self, expr: str): return _compute(expr, self.state)
    def _check(self, cond: str):  return _check(cond, self.state)
    def _grounding_score(self, text: str, facts: dict) -> float:
        # Example: enforce that any ticker in text belongs to facts['top3'] if that exists
```

```python
        import re
        if "top3" in facts:
            mentions = set(re.findall(r"\b[A-Z]{1,5}\b", text))
            target = set(facts["top3"])
            if not mentions: return 0.5  # neutral
            ok = all(m in target for m in mentions if m.isupper())
            return 1.0 if ok else 0.0
        return 0.5
```

- This adheres to the **BaseTextEnv** interface (returning `BaseTextEnvStepOutput`) and integrates **ToolGroups** the same way the docs show. (skyrl.readthedocs.io)
- If you choose to use tags (`<answer>`), add `generator.sampling_params.stop='["</tool>", "</answer>"]'` in your training configs, same as the **Search** example. (skyrl.readthedocs.io)
- If you prefer **JSON-only** actions, you can skip stop strings.

C) Sample dataset items (now with `final_reference` + `judge_rubric`)

Put these in `data/processed/train_llm.json` along with your existing records; they use the same shape your environment expects:

```
{
  "data_source": "synthetic/llm",
  "env_class": "MCPToolEnv",
  "prompt": [
    { "role": "system", "content": "You have tools DuckDuckGo, yahoo_finance, python_execu
    { "role": "user", "content": "Find top-3 NASDAQ-100 gainers today, collect 5 headlines
  ],
  "reward_spec": {
    "method": "rule",
    "ground_truth": {
      "task_id": "nasdaq100_neg_digest_v2",
      "complexity": "complex",
      "max_turns": 12,
      "limits": { "max_servers": 3, "max_tools": 10 },
      "tool_sequence": [
        { "step": 1, "server": "DuckDuckGo", "tool": "search", "params": { "query": "NASDA
        { "step": 2, "server": "DuckDuckGo", "tool": "fetch_content", "params": { "url": "
        { "step": 3, "server": "yahoo_finance", "tool": "get_yfinance_price_history", "param
        { "step": 4, "server": "python_execution", "tool": "python_execution", "params": {
        { "step": 5, "server": "DuckDuckGo", "tool": "search", "params": { "query": "${top
        { "step": 6, "server": "DuckDuckGo", "tool": "search", "params": { "query": "${top
        { "step": 7, "server": "DuckDuckGo", "tool": "search", "params": { "query": "${top
        { "step": 8, "server": "python_execution", "tool": "python_execution", "params": {
        { "step": 9, "server": "python_execution", "tool": "python_execution", "params": {
        { "step": 10, "server": "slack", "tool": "send_slack_message", "params": { "channe
      ],
      "analysis_rubric": {
        "steps": [],
        "final_answer_requirements": {
          "format": "markdown",
          "must_include": ["top3", "neg_titles"],
          "grounded_from": ["top3", "title_sentiment_map"],
          "quality_criteria": ["relevant headlines", "no hallucinated tickers", "concise <
        }
      },
      "final_reference": {
        "answer_text": "Top-3: NVDA, AMD, META. Negative headlines: '…', '…'.",
        "facts": {"top3": ["NVDA","AMD","META"], "neg_titles": ["…","…"]},
        "citations": {"top3": [4], "neg_titles": [8,9]}
```

```
      },
      "judge_rubric": {
        "weights": { "coverage": 0.35, "grounding": 0.4, "clarity": 0.15, "safety": 0.10
        "target_length_range": [40, 140],
        "schema": {
          "type": "object",
          "properties": {
            "coverage": {"type":"number","minimum":0,"maximum":1},
            "grounding":{"type":"number","minimum":0,"maximum":1},
            "clarity":  {"type":"number","minimum":0,"maximum":1},
            "safety":   {"type":"number","minimum":0,"maximum":1},
            "total":    {"type":"number","minimum":0,"maximum":1}
          },
          "required": ["coverage","grounding","clarity","safety","total"]
        }
      }
    }
  }
}
```

This conforms to SkyRL's dataset contract (compact prompt + env_class + reward_spec), leaving your **long-horizon supervision** under `ground_truth`. (skyrl.readthedocs.io)

D) Training knobs to make multi-turn + final answer work smoothly

- If you adopt **tag-style** final answers or tool calls (e.g., `<answer>…</answer>`), set **stop strings** in generator sampling the same way the **Search** example does: `generator.sampling_params.stop='["</tool>", "</answer>"]'` (or `"</search>"` / `"</answer>"`). (skyrl.readthedocs.io)
- Keep **`generator.async_engine=true`** and **`generator.batched=false`** for **multi-turn** async rollouts (again per Search). (skyrl.readthedocs.io)
- Judge model and policy model can be separate; see **LLM-as-a-Judge** example for using an OpenAI model for the judge while training a Qwen policy. (skyrl.readthedocs.io)

E) Why this is mathematically sound (GRPO/PPO)

You’re maximizing expected return (\mathbb{E}_\pi[\sum_t r_t]) where:

- (r_1,\dots,r_K) = **shaped instrumented rewards** (tool choice, param binding, extraction success, computations, accept_if), and
- (r_{K+1} = \lambda_h \cdot r^{\text{heur}}(y_T) + \lambda_j \cdot r^{\text{LAJ}}(y_T, y^)) *is the* ***terminal reward*** *for the final text, with (y_T) the policy’s output and (y^*) the reference.
  LAJ returns a **numeric** score (0–1), so it slots directly into the return — this is exactly what the **SkyRL LLM-as-a-Judge** example does (for GSM8K it’s binary, we generalize to graded). (skyrl.readthedocs.io)

F) Sanity tests you can add right now

- **Unit test**: pass a fake step with `extract=["content"]` → ensure `self.state["content"]` updated and reward includes `extract` term.
- **Final answer test**: with `facts = {"top3": ["NVDA"]}`, final text `"Top-3: NVDA"` → coverage=1, grounding=1 → reward close to weights sum.
- **Judge smoke**: mock the OpenAI response to return `{"coverage":1,"grounding":1,"clarity":1,"safety":1,"total":1}` and ensure reward path works offline.

G) Where this maps to the SkyRL documentation

- **Create env** with **BaseTextEnv**, define `step` that parses tool calls and final answers → “Creating a New Environment” tutorial (and the `BaseTextEnv` API excerpt). (skyrl.readthedocs.io)
- **Use ToolGroups** and `_execute_tool` → “Using Tools in SkyRL-Gym” (core concepts, ToolGroup, env integration, action parsing). (skyrl.readthedocs.io)
- **Multi-turn with final answer** and **stop strings** → “Multi-Turn RL for Search” example

and run config. (skyrl.readthedocs.io)

- **Judge integration** → "LLM as a Judge for GSM8K" example (structure, env calling judge model). (skyrl.readthedocs.io)
- **Dataset contract** → "Dataset Preparation" (data_source, prompt, env_class, reward_spec). (skyrl.readthedocs.io)

TL;DR implementation checklist

1. **Generator** (`generate_with_llm.py`)
   - ☑ Extend schema with `final_answer_requirements` + `judge_rubric` .
   - ☑ Execute plan → `ExecOut(state, steps)` via MCP tools.
   - ☑ Compose `final_reference` with an LLM (or template) grounded in `state` .
   - ☑ Write `reward_spec.ground_truth` including `final_reference` + `judge_rubric` .
2. **Environment** (`mcp_tool_env.py`)
   - ☑ Parse action as tool or final; call tools via ToolGroups.
   - ☑ Implement **per-turn rubric scoring** and **final LAJ scoring** (with structured outputs).
   - ☑ Return `BaseTextEnvStepOutput` each turn.
3. **Training config**
   - ☑ If using tags, set `generator.sampling_params.stop` .
   - ☑ Enable async multi-turn rollouts as in Search.

Once you wire the ToolGroups to your MCP servers, this will let you train **long-horizon, multi-tool research agents** whose **final textual answers** are graded both **heuristically** and by an **LLM judge**, entirely within SkyRL's standard interfaces. (skyrl.readthedocs.io)

If you want me to turn these snippets into line-exact diffs against your current files, paste the full current contents of `generate_with_llm.py` and `mcp_tool_env.py` (without `...`) and I'll produce a ready-to-apply patch.