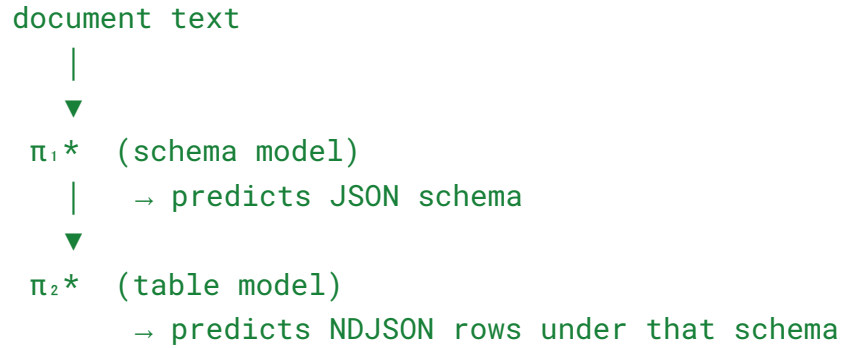


# 0. The Whole Thing in One Picture

**Runtime pipeline (what you ultimately want):**



Final models:

- $\pi_1^*$  = Stage-1 schema model after pipeline RL  $\rightarrow \pi_1^r$ -pipe
- $\pi_2^*$  = Stage-2 table model after pipeline RL  $\rightarrow \pi_2^r$ -pipe

---

**Training phases (how you get there):**

1. **Phase A – SFT (done)**

- $\pi_1^s$  : text  $\rightarrow$  schema
- $\pi_2^s$  : text + schema  $\rightarrow$  table

2. **Phase B – Local RL** (each model alone)

- B1:  $\pi_1^s \rightarrow \pi_1^r$ -**local** (schema reward only)
- B2:  $\pi_2^s \rightarrow \pi_2^r$ -**local** (table reward only, with gold schemas)

3. **Phase C – Pipeline RL** (models help each other)

- C1:  $\pi_1^r$ -local  $\rightarrow \pi_1^r$ -**pipe** (uses  $\pi_2^r$ -local inside reward)

- C2:  $\pi_2^r$ -local  $\rightarrow$   **$\pi_2^r$ -pipe** (trained on schemas from  $\pi_1^r$ -pipe)

#### 4. Phase D – Eval & looping

- Decide if you stop after 1 pipeline loop or do a second.

Stack:

- **VERL** – training orchestrator (SFT + RL)
  - **FSDP** – how VERL trains big LLMs across GPUs
  - **vLLM** – optional fast generator for RL sampling
- 

# 1. Building Blocks (Shared Across All Phases)

## 1.1 Golden data (what you supervise / reward against)

Folder:

```
/scratch/musmanme/common_golden_schema_table/  
common_train_schemas.jsonl  
common_validation_schemas.jsonl  
common_test_schemas.jsonl
```

Each line (simplified):

```
{  
  "id": "...",  
  "split": "train",  
  "text": "A coffee shop ... Blue Spice.",  
  "schema": { "table_id": "...", "columns": [ { "name": "name" }, ... ]  
},
```

```
"table": { "table_id": "...", "columns": [...], "data": [[...],
...] }
}
```

This is:

- **Ground truth schemas** for Stage-1.
  - **Ground truth tables** for Stage-2 and pipeline evaluation.
- 

## 1.2 Prompts (how you talk to the models)

### Stage-1 prompt (schema induction, $\pi_1$ )

Built by your SFT Stage-1 builder:

```
<|policy|>
[POLICY]
- rules about naming, evidence, types, etc.

<|metadata|>
[METADATA]
{ "language": ..., "structure_candidates": [...], ... }

<|document|>
[DOCUMENT]
<raw text>

<|task|>
[TASK]
You induce minimal JSON schemas...
```

- **All RL for  $\pi_1$  uses this exact same user content.**
-

## Stage-2 prompt (table extraction, $\pi_2$ )

Built by `stage2_make_messages_min.py`:

[POLICY]

- Use only facts in the document...
- Output exactly the columns in [SCHEMA]...

[SCHEMA]

```
{"schema_id": "...", "structure": "...", "fields": [{ "name": "name",  
... }, ...]}
```

<|document|>

[DOCUMENT]

<raw text>

[OUTPUT\_FORMAT]

Emit ONLY JSONL, one JSON object per row...

[TASK]

Fill the table under [SCHEMA]...

- **All RL for  $\pi_2$**  uses this too (sometimes with gold schema, sometimes with predicted schema).

---

## 1.3 Metrics (what your rewards & evals care about)

### Schema metrics (Stage-1)

From `src/common/schema_utils.py`:

- `schema_prf_from_text(pred_schema_text, gold_schema_dict) → precision, recall, f1, valid_json.`

These drive:

- **Stage-1 local reward**
  - Part of **pipeline schema reward**
  - **Stage-1 eval** on test set
- 

## Table metrics (Stage-2)

You'll implement in `src/common/table_utils.py`:

- `parse_ndjson(pred_text)` → list of row dicts
- `gold_table_to_rows(gold_table)` → row dicts
- `cell_accuracy(pred_rows, gold_rows)` → fraction of correctly matched cells

These drive:

- **Stage-2 local reward**
  - Part of **pipeline schema reward** (downstream table quality)
  - **Stage-2 & pipeline eval**
- 

## 1.4 Training stack: VERL + FSDP + vLLM + GRPO

### VERL

- Handles:
  - reading your Parquet datasets
  - launching `torchrun` with multiple GPUs
  - wrapping models in **FSDP**

- looping over steps/epochs
  - calling your **reward function** during RL
- For SFT you already use:
  - `verl.trainer.fsdp_sft_trainer`
- For RL you'll use VERL's **RL trainer** (PPO/GRPO flavor):
  - you point it to:
    - actor model path ( $\pi_1$  or  $\pi_2$ )
    - reference model path ( $\pi_1^s$  or  $\pi_2^s$  /  $\pi_1^r$ -local)
    - Parquet RL dataset
    - reward callback (Python function or class)

Exact trainer class/module can vary by VERL version; the pattern is the same.

---

## FSDP (Fully Sharded Data Parallel)

- FSDP splits the model across your GPUs.
- In VERL config you already use things like:
  - `model.strategy=fsdp`
  - `model.fsdp_config.model_dtype=bfloat16`
  - `trainer.n_gpus_per_node=2`
- You **reuse this** in RL:
  - same shard config
  - same dtype

- just a different trainer (RL instead of SFT).

No extra FSDP code needed: you only configure it.

---

## vLLM

- vLLM = optimized text generation engine.
- RL needs lots of “**generate + logprobs**” calls.
- You have two options:

### Option 1 – Start simple:

- Use the **HF model inside VERL** for generation during RL.
- Simpler, but slower.
- Good for debugging & first full pipeline.

### Option 2 – Optimize later:

- Run vLLM server with the current actor weights.
- Have the RL trainer call vLLM to:
  - sample k responses per prompt
  - optionally return token logprobs
- Periodically sync RL weights from training to vLLM.

Either way, conceptually:

VERL RL trainer

```
|  
├ FSDP-wrapped actor model (for losses/backprop)  
└ vLLM / HF model (for generating rollouts)
```

---

## GRPO (how the policy is updated)

For each prompt  $x$ :

1. Sample  **$k$  responses** from current policy  $\pi$ :  $y_1..y_k$ .
2. Compute rewards  $r_1..r_k$  via your custom reward function.
3. Compute mean reward  $\mu = (1/k) \sum r_i$ .
4. Advantage per response:  $A_i = r_i - \mu$ .

Loss ~ average over samples:

$$\text{loss} \approx - \sum A_i * \log \pi(y_i | x) + \text{kl\_coef} * \text{KL}(\pi || \pi_{\text{ref}})$$

5.
  - $\pi_{\text{ref}}$  = frozen reference model (usually SFT).
  - GRPO doesn't need a separate value head; the "group mean" is your baseline.

VERL handles the math; you just provide:

- how to generate
  - how to compute reward per response
- 

## 2. Phase A – SFT (Quick)

You've already done this, but it anchors everything.

### A1. Stage-1 SFT ( $\pi_1^s$ )

- Input messages:
  - system = schema instructions

- user = Stage-1 prompt (<|policy|> ... <|task|>)
  - assistant = gold schema JSON from `build_assistant_schema`.
- Trained with:
  - `train_sft_stage_one.py`
  - VERL + FSDP + FlashAttention-2.
- Output:
  - HF model dir:  $\pi_1^s$

## A2. Stage-2 SFT ( $\pi_2^s$ )

- Input messages from `stage2_make_messages_min.py`:
  - system = “You convert documents into tabular data...”
  - user = [POLICY] + [SCHEMA] + [DOCUMENT] + ...
  - assistant = gold NDJSON rows.
- Trained with:
  - `train_sft_stage_two.py`
- Output:
  - HF model dir:  $\pi_2^s$

You'll use  $\pi_1^s$ ,  $\pi_2^s$  as:

- RL initialization checkpoints (actors)
  - references for KL terms.
-

# 3. Phase B1 – Stage-1 Local RL ( $\pi_1^r$ -local)

## B1.1 What

Train  $\pi_1$  to output **better schemas**, using **only schema reward**.

text + Stage-1 prompt

- $\pi_1$
- predicted schema JSON
- schema reward vs gold

- Start from  $\pi_1^s$ .
  - End with  $\pi_1^r$ -local.
- 

## B1.2 Why

$\pi_1^s$ :

- outputs correct-looking JSON most of the time
- captures most fields, but:
  - may add hallucinated fields
  - may miss fields
  - sometimes invalid JSON

Stage-1 local RL:

- maximizes field F1
  - penalizes invalid JSON
  - tightens behavior **before** worrying about tables.
-

## B1.3 Where (data & code)

- Input Parquet:  
    `data/rl_stage1_schema/train_prompts.parquet`  
    (created by `scripts/stage1_build_schema_rl_parquet.py`)

Each row:

```
id
prompt          # Stage-1 user content
gold_schema_json # canonical schema string
```

- Reward code you already have:  
    `src/rl_stage1_schema/schema_reward_core.py`

You add:

- `scripts/train_rl_stage1_local.py` → uses VERL RL trainer.

---

## B1.4 How (implementation)

**Reward function (inside your reward callback):**

```
cfg = SchemaRewardConfig(
    f1_weight=1.0,
    precision_weight=0.0,
    recall_weight=0.0,
    invalid_json_penalty=-1.0,
    min_reward=-1.0,
    max_reward=1.0,
)

def schema_reward(pred_schema_text: str, gold_schema_json: str) ->
float:
    gold_schema = json.loads(gold_schema_json)
```

```
    info = compute_schema_reward_single(pred_schema_text, gold_schema,
    cfg)
    return info["reward"]
```

### RL training with VERL (conceptual):

- Config in yaml / CLI overrides:
  - model:
    - `partial_pretrain = path/to/pi1_sft ( $\pi_1^s$ )`
    - `strategy = fsdp`
  - data:
    - `train_files = data/rl_stage1_schema/train_prompts.parquet`
    - `prompt_key = "prompt"`
  - rl:
    - `group_size = 4` (k samples per prompt)
    - `kl_coef  $\approx$  0.01` (tune)
    - `reward_callback = your_schema_reward_class`

Inside your reward callback:

```
class SchemaRewardFn:
    def __init__(self, cfg):
        self.cfg = cfg

    def __call__(self, batch_prompts, batch_responses,
    batch_metadata):
        # batch_prompts: list[str]
        # batch_responses: list[str] (predicted schemas)
```

```

# batch_metadata: per-example gold_schema_json
rewards = []
for resp, meta in zip(batch_responses, batch_metadata):
    gold_schema_json = meta["gold_schema_json"]
    r = schema_reward(resp, gold_schema_json)
    rewards.append(r)
return rewards # list[float]

```

VERL:

- sends prompts to the actor (FSDP model) for generation
- calls `SchemaRewardFn` to get rewards
- runs GRPO updates.

**When to stop B1:**

- Have a validation parquet or join with validation JSONL.
- Every N RL steps (e.g. every 200):
  - sample a fixed validation set
  - compute:
    - mean schema F1
    - JSON validity rate
- Keep best checkpoint by val F1.
- Stop when no improvement for several evals (early stopping).

Now you have  $\pi_{1^r}$ -local.

---

## 4. Phase B2 – Stage-2 Local RL ( $\pi_{2^r}$ -local)

## B2.1 What

Train  $\pi_2$  to make **better tables**, assuming **gold schemas**.

Stage-2 prompt (gold schema + text)

- $\pi_2$
- predicted NDJSON
- table reward vs gold table

- Start from  $\pi_2^S$ .
  - End with  $\pi_2^{\text{r-local}}$ .
- 

## B2.2 Why

$\pi_2^S$ :

- already respects schema format
- matches many cells
- but may:
  - miss some rows
  - hallucinate rows
  - muddle some values

Stage-2 local RL focuses purely on **table accuracy given perfect schema**.

---

## B2.3 Where

Create Stage-2 RL data:

- Script: `scripts/stage2_build_table_rl_parquet.py`

- Output:  
`data/rl_stage2_table/train_prompts.parquet`  
`data/rl_stage2_table/validation_prompts.parquet`

Each row:

```
id
prompt          # Stage-2 user message (from stage2_make_messages_min
style)
gold_table_json # full table object
```

You add:

- `src/common/table_utils.py`
- `src/rl_stage2_table/table_reward_core.py`
- `scripts/train_rl_stage2_local.py`

---

## B2.4 How

**table\_utils.py (core pieces):**

```
def parse_ndjson(text: str) -> list[dict]:
    rows = []
    for line in text.splitlines():
        line = line.strip()
        if not line:
            continue
        rows.append(json.loads(line))
    return rows

def gold_table_to_rows(gold_table: dict) -> list[dict]:
    cols = [c["name"] for c in gold_table.get("columns", [])]
    out = []
    for data_row in gold_table.get("data", []):
```

```

        obj = {}
        for i, col in enumerate(cols):
            val = "" if i >= len(data_row) or data_row[i] is None else
str(data_row[i]).strip()
            obj[col] = val
        out.append(obj)
    return out

def cell_accuracy(pred_rows: list[dict], gold_rows: list[dict]) ->
float:
    n = max(len(pred_rows), len(gold_rows))
    if n == 0:
        return 1.0
    correct, total = 0, 0
    for i in range(n):
        pr = pred_rows[i] if i < len(pred_rows) else {}
        gr = gold_rows[i] if i < len(gold_rows) else {}
        keys = set(pr.keys()) | set(gr.keys())
        for k in keys:
            total += 1
            if pr.get(k, "") == gr.get(k, ""):
                correct += 1
    return correct / max(1, total)

```

#### **table\_reward\_core.py:**

```

@dataclass
class TableRewardConfig:
    invalid_json_penalty: float = -1.0
    min_reward: float = -1.0
    max_reward: float = 1.0

def compute_table_reward_single(pred_ndjson_text: str,
                                gold_table: dict,
                                cfg: TableRewardConfig) -> float:
    try:
        pred_rows = parse_ndjson(pred_ndjson_text)
    except Exception:

```

```

    return cfg.invalid_json_penalty

gold_rows = gold_table_to_rows(gold_table)
acc = cell_accuracy(pred_rows, gold_rows)
return cfg.min_reward + (cfg.max_reward - cfg.min_reward) * acc

```

### VERL RL setup (Stage-2):

- Actor:  $\pi_2$  (start from  $\pi_2^s$ ).
- Reference:  $\pi_2^s$  (frozen).
- Data: Stage-2 RL Parquet.
- Reward callback uses `compute_table_reward_single`.

Rest is the same GRPO pattern as Stage-1.

### Stopping B2:

- Validation metrics:
  - table cell accuracy
  - maybe row EM
- Early stopping with patience, keep best  $\pi_2^r$ -local.

---

## 5. Phase C1 – Pipeline RL-1 ( $\pi_1^r$ -pipe)

### C1.1 What

Train  $\pi_1$  so that its schemas:

- are good vs gold **and**
- produce **good tables** when  $\pi_2$  uses them.

text + Stage-1 prompt

→  $\pi_1$

→ predicted schema  $\hat{s}$

(text,  $\hat{s}$ )

→  $\pi_2^r$ -local (frozen)

→ predicted NDJSON  $\hat{t}$

reward =  $\alpha$  \* schema\_reward( $\hat{s}$ , gold\_schema) + (1- $\alpha$ ) \* table\_reward( $\hat{t}$ , gold\_table)

- Start from:  $\pi_1^r$ -local.
- Frozen executor:  $\pi_2^r$ -local.
- End with:  $\pi_1^r$ -pipe.

---

## C1.2 Why

Local RL made schemas match gold better, but:

- Some schemas might be “technically correct” but awkward for  $\pi_2$ .
- Maybe naming + structure choices make extraction harder.

Pipeline RL-1 says:

“Prefer schemas that **actually maximize downstream table accuracy**.”

---

## C1.3 Where

Build a **pipeline RL dataset** for Stage-1:

- `scripts/build_stage1_pipeline_rl_parquet.py`
- Output:  
`data/rl_stage1_schema/pipeline_train.parquet`

`data/rl_stage1_schema/pipeline_validation.parquet`

Each row:

```
id
prompt          # Stage-1 user content
gold_schema_json # canonical schema
text            # original document
gold_table_json  # canonical table
```

Add:

- `src/rl_stage1_schema/schema_pipeline_reward.py`
- `scripts/train_rl_stage1_pipeline.py`

---

## C1.4 How

**Pipeline reward function:**

```
@dataclass
class PipelineSchemaRewardConfig:
    local_schema_weight: float = 0.5
    pipeline_table_weight: float = 0.5
    schema_cfg: SchemaRewardConfig = SchemaRewardConfig()
    table_cfg: TableRewardConfig = TableRewardConfig()
    min_reward: float = -1.0
    max_reward: float = 1.0

def compute_pipeline_schema_reward(pred_schema_text: str,
                                   ex: dict,
                                   table_model,
                                   cfg: PipelineSchemaRewardConfig) ->
float:
    # 1. schema part
    gold_schema = json.loads(ex["gold_schema_json"])
```

```

    schema_info = compute_schema_reward_single(pred_schema_text,
gold_schema, cfg.schema_cfg)
    r_schema = schema_info["reward"]

    # 2. table part (run  $\pi_2^r$ -local under predicted schema)
    try:
        pred_schema_json = json.loads(pred_schema_text)
        user_prompt_stage2 = build_user_message(ex["text"],
pred_schema_json)
        pred_ndjson = table_model.generate(user_prompt_stage2,
max_new_tokens=...)
        gold_table = json.loads(ex["gold_table_json"])
        r_table = compute_table_reward_single(pred_ndjson, gold_table,
cfg.table_cfg)
    except Exception:
        r_table = cfg.table_cfg.invalid_json_penalty

    # 3. combine
    r = cfg.local_schema_weight * r_schema + cfg.pipeline_table_weight
* r_table
    return max(cfg.min_reward, min(cfg.max_reward, r))

```

### Stack:

- Actor:  $\pi_1$  (start from  $\pi_1^r$ -local).
- Reference:  $\pi_1^s$  or  $\pi_1^r$ -local.
- Frozen  $\pi_2$ :  $\pi_2^r$ -local, loaded once; used in reward function:
  - either directly via HF
  - or via vLLM server for speed.

### Stopping C1:

Validation metric should reflect pipeline:

- schema F1

- downstream table accuracy when applying  $\pi_2^r$ -local to  $\pi_1$ 's schemas

Early stop as before, keep best  $\pi_1^r$ -pipe.

---

## 6. Phase C2 – Pipeline RL-2 ( $\pi_2^r$ -pipe)

### C2.1 What

Train  $\pi_2$  to be **robust to real schemas** from  $\pi_1^r$ -pipe.

text

- $\pi_1^r$ -pipe (frozen)
- predicted schema  $\hat{s}$

(text,  $\hat{s}$ )

- $\pi_2$
- predicted NDJSON  $\hat{t}$

reward = table\_reward( $\hat{t}$ , gold\_table)

- Start from:  $\pi_2^r$ -local.
  - Use schemas from:  $\pi_1^r$ -pipe.
  - End with:  $\pi_2^r$ -pipe.
- 

### C2.2 Why

In the real pipeline:

text →  $\pi_1^r$ -pipe →  $\hat{s}$  →  $\pi_2$  →  $\hat{t}$

Schemas are:

- not perfectly gold
- might miss a column
- might have slight name changes

Pipeline RL-2 teaches  $\pi_2$  how to:

- still extract the right values from text
- not crash on minor schema imperfectness.

---

## C2.3 Where

Precompute schemas from  $\pi_1$ -pipe:

- `scripts/pipeline_precompute_schemas_pi1.py`
- Output:  
`data/pipeline/pi1_schemas_train.parquet`  
`data/pipeline/pi1_schemas_validation.parquet`

Each row:

```
id
text
pred_schema_text    # schema from  $\pi_1$ -pipe
gold_table_json
```

Train with:

- `scripts/train_rl_stage2_pipeline.py`

---

## C2.4 How

### Precompute schemas:

```
pi1_pipe = load_hf_model(pi1_pipeline_path)
rows = []

for ex in read_jsonl(common_train_schemas.jsonl):
    s_prompt = build_user_content(ex["text"])
    pred_schema = pi1_pipe.generate(s_prompt,
max_new_tokens=MAX_SCHEMA_TOKENS)
    rows.append({
        "id": ex["id"],
        "text": ex["text"],
        "pred_schema_text": pred_schema,
        "gold_table_json": json.dumps(ex["table"], ensure_ascii=False)
    })

write_parquet(rows, "data/pipeline/pi1_schemas_train.parquet")
```

### RL training:

- Actor:  $\pi_2$  (start from  $\pi_2^{\text{r-local}}$ ).
- Reference:  $\pi_2^s$ .
- Data: pipeline schemas Parquet.

### Reward callback:

```
def pipeline_table_reward(pred_ndjson_text: str, ex: dict, cfg:
TableRewardConfig) -> float:
    gold_table = json.loads(ex["gold_table_json"])
    return compute_table_reward_single(pred_ndjson_text, gold_table,
cfg)
```

### Prompt building inside trainer:

```
schema_json = json.loads(ex["pred_schema_text"])
user_prompt = build_user_message(ex["text"], schema_json)
pred_ndjson = pi2_actor.generate(user_prompt)
```

```
r = pipeline_table_reward(pred_ndjson, ex, cfg)
```

**Stack:**

- VERL RL trainer.
- FSDP on  $\pi_2$  actor.
- vLLM optional for sampling.

**Stop C2:**

- Validation with:
  - text +  $\pi_1$ -pipe schemas
  - reward = table accuracy vs gold.
- Early stopping by best metric.

Now you have  $\pi_2$ -pipe.

---

## 7. Phase D – Evaluation & How Many Loops

### D1. Inference scripts (quick sketch)

**Stage-1 inference ( $\pi_1$ ):**

```
for ex in read_jsonl(input):
    s_prompt = build_user_content(ex["text"])
    pred_schema = pi1_star.generate(s_prompt)
    write({
        "id": ex["id"],
        "text": ex["text"],
        "pred_schema_text": pred_schema
```

```
    })
```

1.

**Stage-2 inference ( $\pi_2^*$ ):**

```
for ex in read_jsonl(input): # has text + schema_text
    schema_json = json.loads(ex["schema_text"])
    t_prompt = build_user_message(ex["text"], schema_json)
    pred_ndjson = pi2_star.generate(t_prompt)
    write({...})
```

2.

**Full pipeline eval:**

```
for ex in read_jsonl(common_test_schemas.jsonl):
    s_prompt = build_user_content(ex["text"])
    pred_schema = pi1_star.generate(s_prompt)

    t_prompt = build_user_message(ex["text"], json.loads(pred_schema))
    pred_ndjson = pi2_star.generate(t_prompt)

    schema_metrics = schema_prf_from_text(pred_schema, ex["schema"])
    table_metrics = table_reward(pred_ndjson, ex["table"])
```

3.

---

## D2. How long to train each phase?

**Inside each RL phase (B1, B2, C1, C2):**

- Use validation-based early stopping:
  - Every `eval_every` steps (say 200):
    - run eval:
      - Stage-1 phases: schema F1 (+ JSON validity).

- Stage-2 phases: table cell accuracy.
  - Pipeline RL-1: maybe weighted combo.
  - Pipeline RL-2: table accuracy with pred schemas.
- Track `best_metric`.
- If no improvement by `min_delta` for `patience` evals:
  - stop, keep best checkpoint.

This is exactly how RLHF-style runs are typically controlled in practice.

---

## D3. How many outer pipeline loops?

Outer loop = full:

- B1:  $\pi_1^s \rightarrow \pi_1^{r\text{-local}}$
- B2:  $\pi_2^s \rightarrow \pi_2^{r\text{-local}}$
- C1:  $\pi_1^{r\text{-local}} \rightarrow \pi_1^{r\text{-pipe}}$
- C2:  $\pi_2^{r\text{-local}} \rightarrow \pi_2^{r\text{-pipe}}$

Do this:

1. **Always complete Loop 1 once.**
2. Evaluate 3 settings on validation/test:
  - SFT baseline:  $\pi_1^s + \pi_2^s$
  - Local RL only:  $\pi_1^{r\text{-local}} + \pi_2^{r\text{-local}}$  (optional ablation)
  - Full pipeline RL:  $\pi_1^{r\text{-pipe}} + \pi_2^{r\text{-pipe}}$
3. If pipeline RL gives a clear boost (it should), ask:

- Is there obvious room left (metrics low)?
- Behavior stable (no weird outputs, KL okay)?

If yes and you have compute, you can do **Loop 2**:

- Freeze  $\pi_2^r$ -pipe as executor in C1.
- Repeat C1+C2 to get  $\pi_1^r$ -pipe-v2,  $\pi_2^r$ -pipe-v2.

Compare Loop 1 vs Loop 2:

- If end-to-end metric (strict EM or table accuracy) improves by  $< \sim 1\%$  or becomes noisy → **stop** and keep Loop-1 or best model.

In most cases:

- **1 outer loop** is plenty.
- **2 loops** max if you see real gains.

---

## 8. Final Checklist (very concrete)

Things you already have:

- ☒ Golden JSONL
- ☒ Stage-1 SFT builder + trainer ( $\pi_1^s$ )
- ☒ Stage-2 SFT builder + trainer ( $\pi_2^s$ )
- ☒ Stage-1 RL Parquet builder (`stage1_build_schema_rl_parquet.py`)
- ☒ `schema_reward_core.py` + `schema_utils.py`

Things to implement:

### 1. Stage-1 RL training script

- `scripts/train_rl_stage1_local.py`
- uses VERL RL trainer + FSDP
- `reward = compute_schema_reward_single`

### 2. Table utils + reward core

- `src/common/table_utils.py`
- `src/rl_stage2_table/table_reward_core.py`

### 3. Stage-2 RL data builder + trainer

- `scripts/stage2_build_table_rl_parquet.py`
- `scripts/train_rl_stage2_local.py`

### 4. Pipeline RL-1 data + reward + trainer

- `scripts/build_stage1_pipeline_rl_parquet.py`
- `src/rl_stage1_schema/schema_pipeline_reward.py`
- `scripts/train_rl_stage1_pipeline.py`
- uses  $\pi_2^r$ -local inside reward.

### 5. Precompute schemas + pipeline RL-2 trainer

- `scripts/pipeline_precompute_schemas_pi1.py`
- `scripts/train_rl_stage2_pipeline.py`

### 6. Inference & eval scripts

- `scripts/infer_stage1_schema.py`

- `scripts/infer_stage2_table.py`
- `scripts/pipeline_eval.py`

Stack is the same throughout:

- VERL orchestrates.
- FSDP shards models.
- HF model or vLLM does generation.
- Your reward code bridges between **text output** and **scalar reward**.